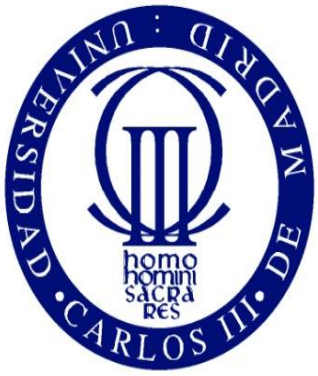


Universidad Carlos III de Madrid

ESCUELA POLITÉCNICA SUPERIOR

DEPARTAMENTO DE TECNOLOGÍA ELECTRÓNICA

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

TRABAJO DE FIN DE GRADO

IMPLEMENTACIÓN DE ANALIZADORES DE PROTOCOLOS DE COMUNICACIONES SPI, I2C

Autor: Salvador Cristina García

Tutor: Mario García Valderas

Fecha de presentación: Octubre de 2014

AGRADECIMIENTOS

Me gustaría dar las gracias a toda mi familia, en especial a mis padres, por haber me apoyado y ayudado a lo largo de mi carrera para lograr mis objetivos. Sin ellos nada hubiera sido posible y me gustaría poder devolverles algún día todo lo que me han dado. Siempre han estado a mi lado, en los momentos buenos y malos y son lo más importante en mi vida.

También querría agradecer todo su apoyo a mi pareja, Laura, la persona que siempre ha estado a mi lado motivándome para seguir adelante y alcanzar mis metas. Ha sido muy importante para mí su ayuda y su actitud, comprendiéndome en cada momento y sabiendo sacar una sonrisa de dónde no la había. Saber que ella está a mi lado es uno de los motivos que me impulsan a seguir avanzando y progresar en la vida.

Por último, querría destacar la importancia en mi progresión de los grandes profesores y compañeros de trabajo que he tenido, gracias a ellos he podido aprender muchos conocimientos técnicos pero también capacidades y aptitudes que me han sido de gran ayuda.

A todos, gracias por vuestra ayuda.

Salvador Cristina García

RESUMEN

El presente proyecto describe el desarrollo realizado, empleado el lenguaje de descripción de hardware VHDL, para diseñar un analizador de protocolos de comunicación SPI e I2C que pueda ser implementado en una FPGA y comprobar que este funciona adecuadamente.

Este proyecto ha sido realizado en colaboración con el departamento de Tecnología Electrónica de la Escuela Politécnica Superior de la Universidad Carlos III de Madrid.

Este documento describe los aspectos generales de dicho proyecto incluyendo los objetivos que desea alcanzar y las posibles soluciones que se han planteado para alcanzarlos. Además, incluye una descripción de los protocolos que son analizados en el diseño y sus principales características.

Posteriormente, el documento detalla el diseño realizado para la solución, especificando la implementación y funcionalidad que posee cada uno de los bloques que han sido diseñados.

También se incluye un apartado en el que se describe la fase de verificación del diseño obtenido, incluyéndose los apartados de pruebas de simulación y pruebas a nivel hardware ejecutadas.

Además han sido incluidos algunos datos referentes al proyecto llevado a cabo como son el detalle de tiempos empleados en cada una de las fases de desarrollo del diseño y el entorno socio-económico donde este puede ser empleado.

Finalmente, pueden consultarse en este documento una serie de anexos que facilitarán la comprensión del diseño y permitirán observar la descripción VHDL realizada para obtener el circuito resultante del diseño.

ÍNDICE DE CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE ECUACIONES

ABREVIATURAS

Abreviatura	Procedencia	Significado
ADC	Analog to Digital Converter	Conversor de Analógico a Digital
ASCII	American Standard Code for Information Interchange	Código Estándar Estadounidense para el Intercambio de Información
ASIC	Aplication Specific Integrated Circuits	Circuitos integrados de Aplicación específica
BCD	Bynary-Coded Decimal	Decimal Codificado en Binario
CS	Chip Selector	Selector de Chip
DAC	Digital to Analog Converter	Conversor de Digital a Analógico
EIA	Electronic Industries Association	Asociación de Industrias Electrónicas
FIFO	First Input First Output	Primero en Entrar Primero en Salir
FPGA	Field Programmable Gate Array	Matriz de Puertas Programables en Campo
FSM	Finite State Machine	Máquina de Estado Finito
I2C	Inter Integrated circuit	Inter Circuitos Integrados
ISE	Integrated Software Environment	Entorno de Desarrollo de Software Integrado
LSB	Less Significant Bit	Bit Menos Significativo
LUT	Look Up Table	Tabla de Consulta
MISO	Master Input Slave Output	Entrada del Maestro Salida del Esclavo
MOSI	Master Output Slave Input	Salida del Maestro Entrada del Esclavo
MSB	Most Significant Bit	Bit Más Significativo
RTL	Register-Transfer Level	Nivel de Transferencia de Registros
SCLK	Serial Clock	Reloj Serie
SoC	System on Chip	Sistema en Chip
SPI	Serial Peripheral Interface	Bus de interfaz de periféricos serie
UART	Universal Asynchronous Receiver-Transmitter	Transmisor-Receptor Asíncrono Universal
USB	Universal Serial Bus	Bus Serie Universal
VHDL	VHSIC (Very High Speed Integrated Circuit) HDL (Hardware Description Language)	Lenguaje de Descripción de Hardware para Circuitos Integrados de Alta Velocidad

1. Introducción

En la actualidad, la mayoría de los sistemas complejos empleados en la industria se componen de un elemento central encargado del control del sistema y diversos periféricos. Estos periféricos aglutinan funciones específicas consiguiendo que se reduzca la carga de trabajo del sistema de control y aumentando la velocidad del conjunto. Además, es habitual la colaboración entre diversos sistemas para lograr un fin común, por lo que es necesario el intercambio de numerosa información entre ellos.

Esta relación entre sistemas se consigue mediante la conexión de todos los elementos implicados a un bus de comunicaciones, medio empleado para realizar las transmisión entre dispositivos. Generalmente este bus consistente en una unión física entre módulos (empleando conectores y cables específicos). También existen conexiones inalámbricas como, por ejemplo, la conexión WIFI empleada por los enrutadores *(routers)* actuales para comunicar con ordenadores, tabletas, teléfonos móviles u otros dispositivos.

Para lograr el funcionamiento óptimo de un sistema completo, es necesario que las transmisiones entre los dispositivos implicados se realicen de forma adecuada para evitar pérdidas de información o el mal funcionamiento de un equipo.

Por estos motivos, las comunicaciones digitales han cobrado un papel importante en los últimos años, avanzado notablemente su complejidad y desarrollándose diversas tecnologías y protocolos. Estas mejoras permiten aglutinar mayor contenido en un único mensaje, aumentar la velocidad de las transmisiones y aportar mayor seguridad en las comunicaciones entre dispositivos.

Comprendida la importancia de las comunicaciones, es necesario poder verificar y depurar las transmisiones realizadas entre elementos del sistema para comprobar entre otros aspectos: la correspondencia del mensaje enviado/recibido con el esperado, verificar el estado de la línea de comunicación, detectar errores en la transmisión o comprobar que la velocidad de transmisión es la adecuada.

Para evaluar todos estos aspectos generalmente se utiliza un analizador de protocolos, elemento capaz de capturar las tramas enviadas o recibidas en la comunicación y extraerlas para su posterior análisis. Estos dispositivos se presentan en diferentes arquitecturas y existen diversos modelos con características y funcionalidades específicas para cada tipo de comunicación.

El presente proyecto se centrará en el desarrollo de un analizador de comunicaciones serie entre dispositivos, que son aquellas que cuentan con una o dos líneas para la comunicación (dependiendo del tipo de protocolo utilizado serán necesarias una línea para la transmisión maestro-esclavo y otra para la transmisión esclavo-maestro o ambas transacciones se realizarán mediante una única línea). El envío de datos se realiza de forma secuencial, enviándose únicamente un dato en cada momento, es decir, la transmisión se realiza bit a bit y no por grupos como en el caso de las conexiones paralelas.

Específicamente, se analizarán las comunicaciones serie síncronas, que son aquellas que emplean una señal de reloj para controlar la transferencia de los datos, de manera que se envíe un bit en cada ciclo del reloj. Esto permite que el transmisor y el receptor se sincronicen

para escribir y leer el dato de la línea de comunicación en el momento adecuado. De esta manera se evita el envío o recepción de datos erróneos y pérdidas en la comunicación.

1.1 Aspectos generales

Este documento define y especifica el desarrollo de un analizador de protocolos de comunicación serie que permitirá la captura de tramas procedentes de los protocolos SPI (*Serial Peripheral Interface*) e I2C *(Inter-Integrated Circuits*).

En primer lugar se definirá el alcance de este proyecto. A continuación, se describirán los objetivos que se desean alcanzar, se identificará el problema que presenta el desarrollo y se describirán sus posibles soluciones. Posteriormente, se hará una introducción al análisis de los protocolos a analizar, detallándose sus características principales. Seguidamente, se detallará el desarrollo de la solución elegida y la tecnología que se va a emplear para conseguirlo. Acabado este punto, se describirán las pruebas realizadas para corroborar el correcto funcionamiento y adecuación del sistema y se mencionarán los recursos empleados para la realización de las pruebas. Finalmente, se extraerá una serie de conclusiones acerca del desarrollo y se expondrán algunas líneas de posibles trabajos futuros.

1.2 Alcance del proyecto

El presente proyecto abarca los siguientes aspectos:

- Implementación de un analizador de protocolos SPI e I2C capaz de capturar las comunicaciones realizadas entre dispositivos conectados al bus.
- Captura de los datos enviados tanto por el sistema maestro como por el esclavo, dando cobertura total a la comunicación entre ambos.
- Detección de fallos en la transmisión como longitud incorrecta del mensaje o corte de la transmisión.
- El caso estudiado se centra en la comunicación entre un maestro y un esclavo, pudiendo capturarse las transmisiones de diversos esclavos en el caso del bus I2C.
- Envío de las tramas capturadas a una estación de análisis, en este caso un ordenador.
- Monitorización en ordenador de los datos recibidos por el analizador.
- Realización de pruebas del sistema tanto a nivel de simulación como a nivel físico empleando un Hardware específico.
- Análisis de resultados una vez alcanzada la solución presentada.

1.3 Objetivos

- **Objetivos generales**

El objetivo principal de este proyecto es diseñar, empleando el lenguaje VHDL, un sistema capaz de registrar los datos transmitidos entre dispositivos conectados a un bus SPI o I2C y extraerlos para su posterior análisis.

- **Objetivos específicos**
  - Comprender las características y el funcionamiento del protocolo SPI
  - Conocer y entender las particularidades del protocolo I2C.

- Implementar un diseño, usando VHDL, capaz de integrarse en otros diseños o funcionar de manera independiente para capturar las tramas de comunicación y facilitar la depuración de las transmisiones.
- Detectar la perdida de mensajes en la captura.
- Capturar la longitud del mensaje para verificar que es la esperada.
- Detectar pérdidas de comunicación entre dispositivos.
- Diseño de un sistema capaz de ser implementado en una FPGA.
- Seleccionar un software para la recepción en el ordenador de los datos capturados.
- Realizar diversas pruebas que verifiquen el correcto funcionamiento del sistema.
- Analizar los resultados obtenidos y definir posibles líneas futuras de trabajo.

1.4 Antecedentes

El protocolo SPI es un estándar de comunicaciones desarrollado por Motorola en torno a 1980 y empleado habitualmente en comunicaciones entre circuitos integrados. Se trata de un bus serie que permite velocidades de transmisión altas y requiere pocos recursos en el circuito por lo que su consumo es bajo. Este tipo de comunicación se emplea para conectar periféricos a un sistema como pueden ser sensores (de temperatura, presión, posición, etc.), ADC y DAC, potenciómetros digitales o memorias.

El protocolo I2C fue desarrollado por Philips semiconductor en 1992 (hoy en día perteneciente al grupo NXP semiconductors) como una solución a la hora de intercambiar información entre circuitos integrados. Este bus permite una configuración multi-maestro por lo que es muy usado en la industria para comunicaciones entre microcontroladores y periféricos de modo que ambos puedan enviar mensajes en cualquier momento actuando como maestro del bus.

En lo que se refiere al análisis de las comunicaciones entre dispositivos, éste surge a la vez que las primeras comunicaciones digitales debido a la necesidad de comprobar el correcto funcionamiento de éstas.

Existen diversos motivos que requieren el análisis de una comunicación como pueden ser: la identificación de un problema surgido en la integración de un equipo, la comprobación de conexiones entre dispositivos, la monitorización de la comunicación o el análisis de los paquetes enviados durante las transacciones.

Para dar solución a todas estas necesidades se han desarrollado diferentes analizadores de protocolos de comunicación. Estos sistemas son capaces de capturar los datos transmitidos por distintos tipos de protocolos para facilitar la depuración de la comunicación y el control de ésta.

En nuestro caso, emplearemos para el desarrollo el lenguaje VHDL, creado en los años 80 y empleado para la descripción de circuitos digitales. Además, VHDL está definido por el IEEE por lo que se considera un estándar y existe numerosa documentación y pautas de diseño sobre él. Este lenguaje permite la descripción comportamental del circuito que se desea diseñar simplificando significativamente el desarrollo del diseño y permitiendo la creación de sistemas más complejos.

El circuito resultante del diseño será implementado en una FPGA, dispositivo programable de reducido tamaño que contiene bloques lógicos los cuales pueden ser programados para diferentes funciones. Estos dispositivos son más lentos que los ASIC pero tienen la ventaja de que pueden ser reprogramados por lo que se pueden emplear para diversas funciones, desde casos muy simples como el comportamiento de una puerta lógica hasta aplicaciones complejas como sistemas de visión por computadora o sistemas de reconocimiento de voz. Las FPGA son utilizadas frecuentemente en el desarrollo de prototipos por su versatilidad y su bajo coste en comparación con la fabricación de un ASIC.

1.5 Identificación del problema

Siempre que se utiliza una comunicación digital entre dos o más dispositivos es necesario verificar que la transmisión de datos entre ellos se realiza de forma adecuada ya que pueden surgir diversos problemas como errores de conexión, contenido erróneo de paquetes, accesos indebidos al bus o falta de transferencias entre otros.

La comunicación puede verificarse a nivel de simulación o a nivel de hardware empleando un osciloscopio para comprobar las transferencias. Sin embargo, este método es trabajoso ya que requiere de la comprobación de numerosas líneas (línea de reloj, línea de datos, selector de chip…) lo que puede convertir la comprobación en un trabajo tedioso, especialmente cuando las tramas enviadas son extensas y están serializadas.

Para dar solución a este problema será necesario desarrollar un sistema que permita monitorizar la comunicación y analizar los datos que se han transmitido facilitando la depuración del componente a tratar.

El elemento desarrollado debe ser capaz de capturar los datos y almacenarlos para poder mostrarlos en un medio que permita la sencilla comprobación de la trama enviada o recibida. Además, debe identificar el dispositivo que está accediendo al bus, la configuración del protocolo y el contenido del mensaje transferido. Por último, la implementación realizada debe identificar funcionamientos incorrectos en la transmisión como son: longitud inadecuada del mensaje o cortes en la transferencia del mensaje.

1.6 Posibles soluciones

Una vez planteado el problema a resolver, existen diferentes vías para solucionarlo:

- Podría emplearse un microcontrolador que cuente con un periférico capaz de capturar las tramas enviadas por el bus y almacenarlas en una memoria RAM. Este periférico podría ser un ADC que muestrease la línea de datos dependiendo del estado de la línea de reloj. Posteriormente el microcontrolador accedería a dicha memoria RAM para obtener el contenido del mensaje y su configuración y mostrarlo en una pantalla LCD acoplada a la placa del microcontrolador como otro periférico.
  Esta solución requeriría el empleo de un microcontrolador con una capacidad de procesamiento elevada y una velocidad alta dado que las transferencias entre dispositivos son muy rápidas, problema que se puede extrapolar a la capacidad de muestreo de los ADC. Además sería necesaria una memoria con alta capacidad para poder almacenar muchos mensajes. Otro inconveniente de esta solución es que la

pantalla LCD solo permitiría la monitorización de mensajes cortos o de solo una parte del mensaje dado su reducido tamaño.

- Otra posibilidad sería utilizar el mismo microcontrolador que en la solución anterior, pero en este caso los mensajes no serían almacenados en una memoria sino que serían enviados desde un puerto serie (ya sea RS-232 o USB) a un ordenador donde los datos serían monitorizados.

  Esta solución requiere que el microcontrolador disponga de un puerto serie y sería necesario configurarlo y manejarlo a una alta velocidad para evitar la pérdida de mensajes ya que el envío por el puerto debería ser más rápido que la captura del siguiente mensaje. Además, habría que diseñar un software en el ordenador capaz de abrir, leer y escribir en el puerto empleado y manipular los datos para monitorizarlos en pantalla por lo que el diseño resultaría más complejo.
- Considerando otras tecnologías, podría emplearse una FPGA para la conexión y captura del bus. La FPGA dispondría de unas entradas conectadas a las líneas del bus que capturarían los mensajes en función de la línea de reloj y el selector de chip (si éste existe). Los mensajes serían almacenados en una memoria FIFO que posteriormente sería leída para enviar únicamente el contenido de los mensajes a través de un puerto serie al ordenador. En el ordenador, un software se encargaría de acceder al bus y gestionar la información para mostrar los mensajes en pantalla de una forma adecuada.

  Esta solución de nuevo requeriría el desarrollo de dos sistemas claramente diferenciados: por un lado habría que diseñar el hardware contenido en la FPGA y además habría que desarrollar un software capaz de gestionar el puerto por el que se reciben los mensajes y manipularlos para mostrarlos en pantalla.
- Una solución más sencilla y versátil sería contener todo el sistema dentro de la FPGA, de manera que fuese ésta la encargada de capturar el mensaje como se ha explicado en la solución previa, almacenarlo en una memoria FIFO para que no se pierdan los datos y posteriormente gestionar el mensaje para enviarlo al ordenador con una estructura clara y que facilite su comprensión. La FPGA se conectaría mediante un puerto serie al ordenador de modo que en éste únicamente sería necesario disponer de un software capaz de leer el puerto serie para poder monitorizar los mensajes en pantalla.

  Esta solución ofrece varias ventajas respecto a las anteriores como son: la posibilidad de implementar únicamente el sistema en una FPGA que se conecte físicamente al bus o, si el dispositivo que gestiona el protocolo ha sido desarrollado en VHDL, incluir el sistema como un módulo más del desarrollo dentro del mismo proyecto facilitando la depuración del gestor del bus. Además, esta posibilidad no requiere el desarrollo de ningún software ya que existen diversos programas, incluso declarados como código abierto y con versiones portables para diferentes sistemas operativos, capaces de leer el puerto serie de un ordenador. Otro motivo para seleccionar esta solución es el hecho de que, dado que los mensajes se almacenan en una memoria antes de ser enviados al ordenador y que en la pantalla pueden visualizarse varios mensajes de manera simultánea, se reduce la posibilidad de perder mensajes y se puede visualizar un conjunto de mensajes a la vez. Esto facilita la depuración del sistema cuando los

mensajes son enviados en grupos de un tamaño específico como por ejemplo la escritura de una estructura en una memoria.

Considerando los motivos expuestos anteriormente, se ha optado por desarrollar la última de las soluciones presentadas, el desarrollo del sistema completo en VHDL que se implementará en una FPGA. El motivo de esta elección es que dicha solución requiere el empleo de un único lenguaje para todo el desarrollo y permite que el sistema pueda ser implementado en FPGA de distintos fabricantes y familias, aumentado su posibilidad de uso y versatilidad. También se ha seleccionado esta implementación por su sencillez de uso en cualquier ordenador, independientemente del sistema operativo utilizado, ya que únicamente requiere la disponibilidad de un puerto serie, ya sea RS-232 o USB (se puede usar un conversor de RS-232 a USB cuyo coste es bajo) y un programa capaz de leer el puerto ya que todo el manejo de la información se realiza dentro de la propia FPGA.

1.7 Marco regulador

Considerando los diversos elementos empleados en el presente proyecto es necesario tener en consideración siguientes aspectos de normativa:

- En lo que se refiere al protocolo SPI, no existe una especificación formal del bus, toda la información restrictiva referente a este bus se encuentra en las especificaciones y hojas de características de los microcontroladores y demás sistemas que lo incluyen. Por poner un ejemplo, como guía se puede consultar la descripción del bloque SPI implementado por Motorola, accesible en la referencia [REF-1] Guía del bloque SPI de Motorola.
- Por el contrario, el bus I2C posee su propia especificación, donde se definen todos los aspectos normativos (características eléctricas y temporales) que deben tenerse en consideración al desarrollar un dispositivo capaz de conectarse a este tipo de bus. Este documento se revisa y actualiza periódicamente y su última versión puede consultarse en la siguiente referencia: [REF-2] Especificación del bus I2C.
- Respecto a la normativa del puerto serie RS-232, existe una norma definida por el EIA donde se especifican todas las características de este tipo de comunicación incluyendo conectores, características eléctricas, características temporales y formatos de transmisión. Esta norma puede consultarse en la referencia [REF-3] Interfaz entre equipos terminales y de comunicación de datos empleando intercambio de datos binarios en serie.

2. Protocolos de comunicación estudiados

En el presente apartado se describirán las características principales de los dos protocolos de comunicación con los que se va a trabajar, el protocolo SPI y el protocolo I2C. Primeramente, se realizará una introducción a cada uno de los buses para, a continuación, describir sus características y señales. Finalmente se detallará el modo de funcionamiento de cada uno de ellos para facilitar la comprensión del diseño realizado para analizar las tramas enviadas.

2.1 Protocolo SPI

El bus de interfaz de periféricos, conocido como bus SPI, es un estándar de comunicaciones empleado generalmente en transferencias entre circuitos integrados.

Este protocolo fue desarrollado en la década de 1980 por Motorola para permitir la comunicación entre diferentes sistemas electrónicos digitales como sus microcontroladores de la serie 68000 con los que surgió la idea.

Se trata de un bus serie síncrono, es decir, emplea una línea de reloj para sincronizar las transmisiones de datos entre dispositivos. Estas transmisiones se realizan mediante dos líneas, una para la transmisión del sistema maestro al esclavo y otra para la transferencia del esclavo al maestro. Dado que la transmisión es serie, se envía un único bit del contenido del mensaje en cada ciclo del reloj. Para comenzar o finalizar una transmisión el bus incluye una línea de selección de chip que permite identificar con que dispositivo se desea realizar el intercambio de información.

2.1.1 Características

Las características principales del bus SPI son:

- Se trata de un bus de 4 hilos
- Es un protocolo maestro-esclavo que permite controlar varios periféricos a través de un único bus.
- Se considera un sistema de maestro único, es decir, un único dispositivo se encarga de generar el reloj que sincroniza la transmisión e indicar el inicio y fin de ésta al esclavo a través de una línea de selección de chip.
- Permite la conexión de varios sistemas esclavos al mismo bus, de forma que el maestro puede intercambiar información con todos ellos empleando las mismas líneas de reloj y datos. Los esclavos fijan su salida de datos en alta impedancia para permitir que otros esclavos accedan al bus sin que se produzcan distorsiones en el mensaje.
- Posee dos parámetros de configuración: CPHA y CPOL, los cuales definen el sincronismo de la transferencia.
- La comunicación es full-duplex, es decir, permite el envío y recepción de información de manera simultánea de manera que los dispositivos pueden actuar a la vez como emisor y receptor de información.
- Permite velocidades de transmisión altas pudiendo alcanzar hasta 12,5 Mbit/s.
- La longitud de un mensaje no está limitada, pudiendo componerse éste de 1 bit hasta longitudes mucho mayores.
- Su implementación hardware requiere pocos recursos, por lo que el consumo energético de este bus es reducido.
- No requiere mecanismo de arbitraje o de respuesta ante fallos ya que la comunicación es constantemente controlada por el dispositivo maestro.
- El direccionamiento o selección de esclavo se realiza mediante una línea dedicada.
- No dispone de señal de asentimiento, es decir, el maestro podría estar enviando información a un esclavo y no existe la posibilidad de saber si éste la está recibiendo de manera adecuada.

2.1.2 Señales del bus SPI

Como ya se ha comentado, este bus emplea 4 señales para realizar las comunicaciones. La conexión entre los dispositivos maestro y esclavo puede verse en la siguiente imagen:

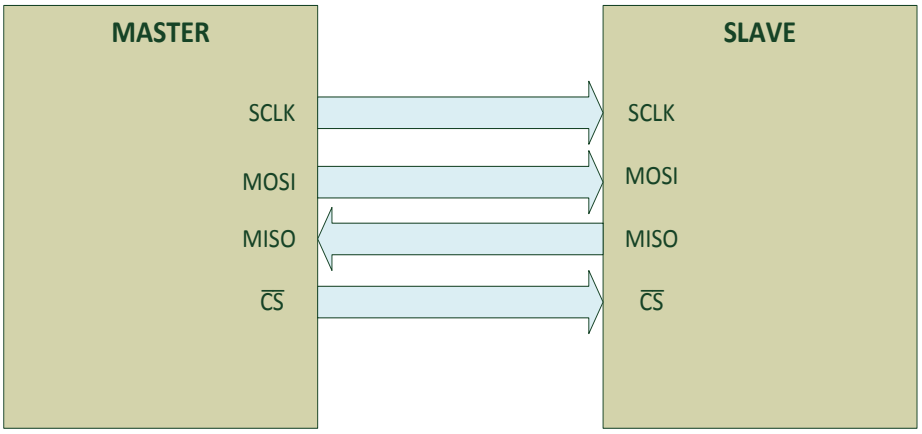


Figura 1: Conexión entre maestro y esclavo SPI

- **SCLK**

Señal de reloj generada por el dispositivo maestro y empleada para sincronizar la transferencia del mensaje. Los datos se enviarán de manera secuencial de forma que cada bit del mensaje permanezca en la línea de datos durante un ciclo de este reloj. La polaridad del reloj puede ser configurada, definiendo el estado en reposo de éste (mientras no se realiza ninguna transmisión el reloj puede estar en nivel alto o bajo).

- **MOSI**

Línea de datos empleada para la transferencia de información del maestro al esclavo. Esta línea se conectará a todos los esclavos presentes en el bus de manera que el maestro pueda comunicar con todos ellos a través de una única línea. Mientras el sistema está en reposo, el maestro pondrá su salida de datos en alta impedancia para permitir otros accesos al bus.

- **MISO**

Conexión de datos utilizada para el envío de mensajes desde el esclavo hacia el maestro. Todos los dispositivos esclavos se conectan a la misma línea de modo que el maestro pueda recibir información de todos ellos a través de una sola entrada. Los esclavos fijan su salida de datos en alta impedancia cuando no están transmitiendo información para que otros esclavos puedan acceder al bus.

- **CS**

Línea de selección de esclavo gestionada por el dispositivo maestro. Cada esclavo posee su propia entrada de selección mientras que el maestro posee una salida para cada uno de los esclavos conectados al bus. El maestro emplea esta línea para indicar al esclavo que se desea realizar con intercambio de información con él.

2.1.3 Descripción del protocolo

A continuación, se describirá el proceso para generar una transmisión entre un dispositivo maestro y un esclavo, definiéndose la velocidad de transmisión de información y los diferentes modos en los que puede realizarse ésta.

- Funcionamiento

Cuando el dispositivo maestro desea realizar una transmisión con un esclavo debe seguir unas pautas para que ésta se realice de forma adecuada:

- ✓ En primer lugar debe seleccionar el esclavo con el que desea intercambiar la información. Para ello debe poner a nivel bajo el selector de chip correspondiente a dicho esclavo. Mientras un esclavo no sea seleccionado, éste mantendrá su salida de datos en alta impedancia para permitir que otros esclavos manejen el bus sin problemas.
- ✓ A continuación, el maestro debe activar el reloj de comunicación (SCLK) generando una señal simétrica (mismo tiempo a nivel alto y bajo) y con una frecuencia adecuada para la velocidad de transmisión deseada.
- ✓ Una vez activado el reloj, el maestro escribirá un bit en la MOSI en cada ciclo de reloj. En el flanco contrario al de escritura, el maestro leerá un bit de la línea MISO.
- ✓ Simultáneamente, el esclavo escribirá otro bit en la línea MISO en cada ciclo, leyendo el bit presente en la línea MOSI en el flanco de reloj contrario a la escritura.
- ✓ Cuando todos los bits hayan sido transmitidos en ambos sentidos, el maestro deshabilitará el reloj de comunicación.
- ✓ Finalmente, el maestro volverá a poner a nivel alto la línea de selección de chip para indicar al esclavo que la transferencia ha finalizado. En este momento el esclavo pondrá su salida de datos en alta impedancia para permitir transferencias de otros esclavos.

La siguiente imagen muestra un ejemplo de transmisión entre los dispositivos maestro y esclavo. Las características temporales de esta transmisión pueden variar en función del modo de transmisión seleccionado, los cuales serán explicados posteriormente.

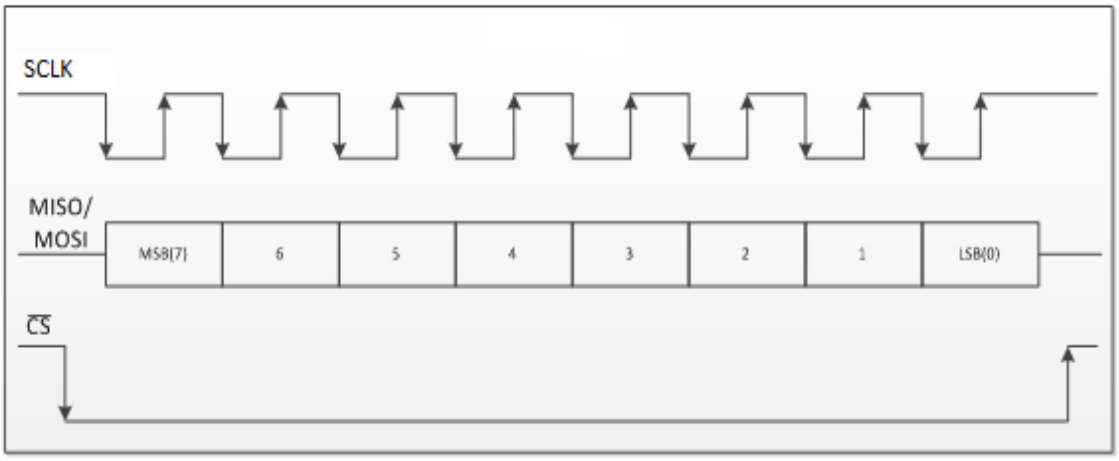


Figura 2: Ejemplo de transmisión SPI

- Velocidad de transmisión (baudios)

La velocidad de transmisión del Interfaz de Periféricos Serie queda definida como la cantidad de bits transferidos en un segundo a través de la línea de datos, es decir, bits/s. Dado que, como hemos comentado anteriormente, la transmisión de cada bit se realiza durante un ciclo del reloj SCLK, la velocidad de transmisión depende de la frecuencia del reloj SCLK como define la siguiente fórmula:

$$Baudios = {}^{1}\Big/_{\frac{1}{F_{SCLK}(Hz)}}$$

Ecuación 1: Velocidad de transmisión bus SPI

Dónde:

- Baudios: Velocidad de transmisión expresada en bits/s
- F_{SCLK} : Frecuencia del reloj de transmisión serie expresada en Hercios.

Dado que este bus no posee una especificación propia, no están definidas velocidades mínimas ni máximas de transmisión, pudiendo estas tomar valores desde 12500 bit/s hasta 12500000 bits/s como refleja la tabla 3.4 del documento [REF-1] Guía del bloque SPI de Motorola.

- Modos de transferencia:

El bus SPI emplea 2 parámetros para definir el momento en el que se escriben o leen las líneas de datos, estos dos parámetros son:

- CPHA: Este parámetro define el retraso respecto al primer flanco del reloj de sincronización.
  - CPHA = 0: No hay retraso. El primer bit se leerá en primer flanco del reloj, lo que implica que el bit debe ser escrito en la línea de datos previo al primer flanco de reloj, tanto en el caso del maestro como del esclavo.
  - CPHA = 1: Debe considerarse un flanco de retraso en la lectura. El primer bit se leerá en segundo flanco de reloj. Debido a esta condición, el primer bit debe escribirse en la línea de datos con el primer flanco del reloj para asegurar que es estable en el segundo flanco, momento en el que se realizará la lectura.
- CPOL: Define el estado de reposo de la línea de reloj SCLK, es decir, el nivel que tendrá la línea de reloj mientras que no se está realizando ninguna transferencia. Este parámetro puede tener 2 valores:
  - CPOL = 0: La línea SCLK permanecerá a nivel bajo mientras no se realicen transferencias.
  - CPOL = 1: La línea del reloj de sincronización se mantendrá a nivel alto cuando no se están efectuando operaciones de transmisión.

Para que el intercambio de información se realice de manera adecuada es necesario que el maestro y el esclavo tengan la misma configuración de estos parámetros. Analizando las posibles combinaciones, surgen 4 posibles modos de funcionamiento.

- Modo 1: CPHA = 0 y CPOL = 0
  En este modo la línea de reloj permanece a nivel bajo en reposo y no se considera retraso respecto al primer flanco de éste. Por ello, cuando se inicia una transmisión con

la bajada de la línea de selección de chip, tanto el maestro como el esclavo deben escribir el primer bit antes de que llegue el primer flanco del reloj para asegurar que la lectura se realiza de forma correcta. La escritura del resto de bits se realizará con los flancos de bajada del reloj para que el valor sea estable durante el tiempo en alto del reloj. El primer bit, y todos los demás, se leerán en los flancos de subida del reloj como refleja la figura 3:

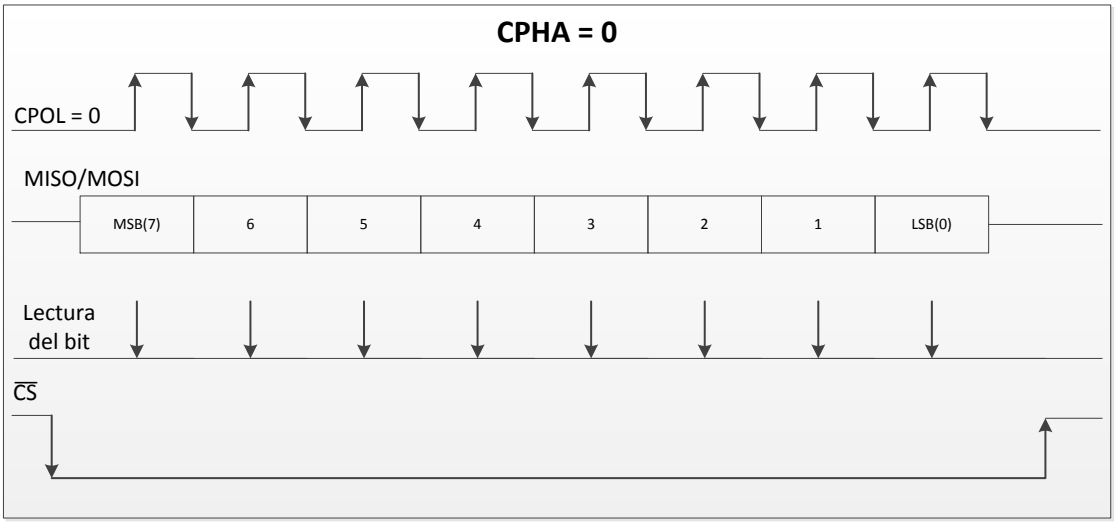


Figura 3: Modo 1 de transmisión SPI

- Modo 2: CPHA = 0 y CPOL = 1
  Este modo se comporta de manera análoga a anterior, con la diferencia del estado de reposo de la línea de reloj que, en esta ocasión, permanecerá a nivel alto mientras que no se efectúen operaciones. De nuevo, el primer bit debe escribirse en la línea de datos antes del primer flanco de reloj, que en este caso será un flanco de bajada. La escritura del resto de bits se realizará con los flancos de subida del reloj asegurando así la estabilidad del dato en el momento de la lectura. Todas operaciones de lectura se realizarán con el flanco de bajada del reloj como puede observarse en la figura 4:

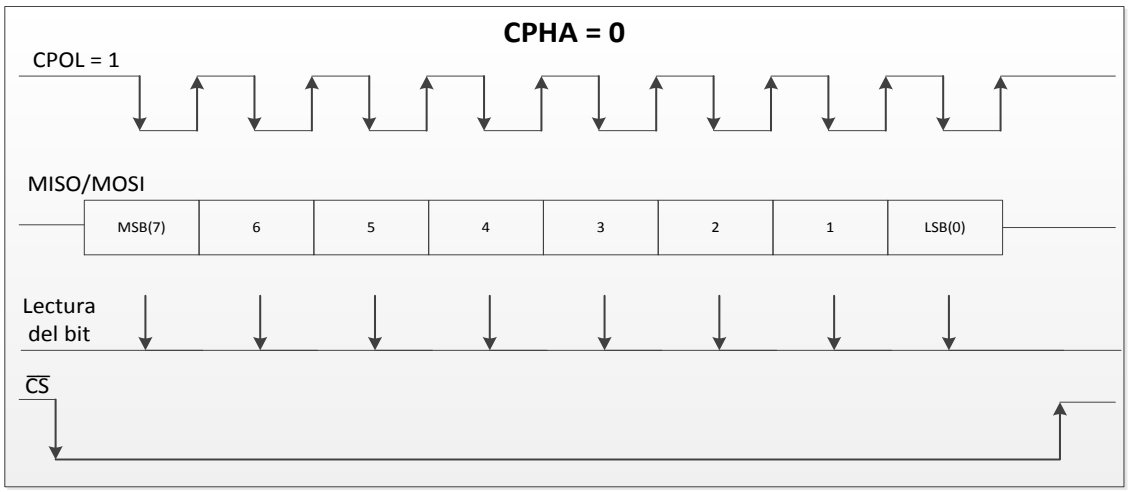


Figura 4: Modo 2 de transmisión SPI

- Modo 3: CPHA = 1 y CPOL = 0

Con esta configuración debe considerarse un retraso respecto a la línea de reloj SCLK. Por ello, la escritura del primer dato en la línea serie se realizará después del primer flanco de reloj, en este caso, flanco ascendente dado el valor del parámetro CPOL. La escritura del resto de bits se realizará de nuevo con los posteriores flancos ascendentes del reloj. En el caso de la lectura, ésta se efectuará en los flancos de bajada del reloj, momento en el que el dato debe ser estable como refleja la siguiente imagen:

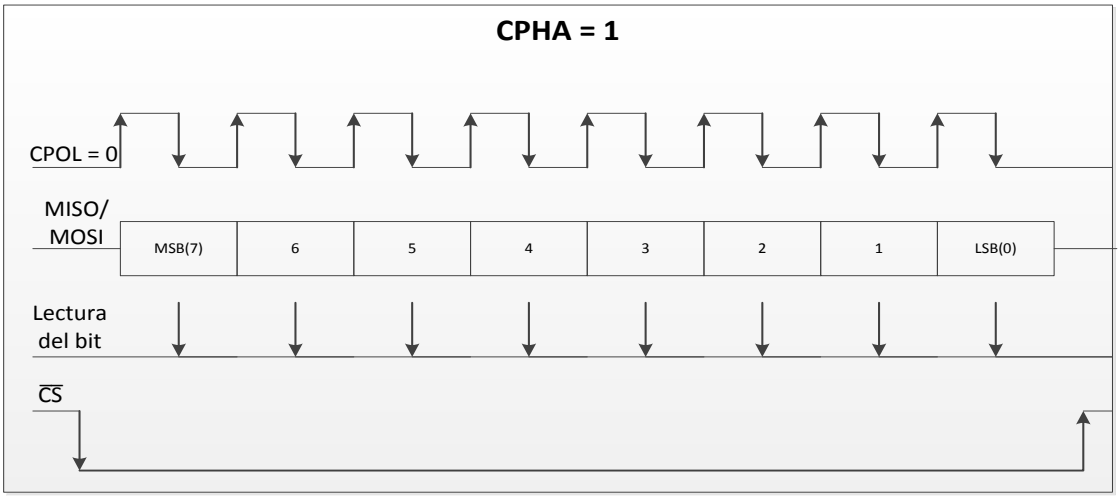


Figura 5: Modo 3 de transmisión SPI

- Modo 4: CPHA = 1 y CPOL = 1

  Este modo es similar al anterior, variando de nuevo el estado de reposo de la línea SCLK, que en este caso quedará fijada a nivel alto durante el reposo. La escritura de los bits en la línea de datos se efectuará con el flanco de bajada del reloj, comenzando en el primer flanco. El proceso de lectura se realizará con los flancos de subida del reloj, de modo que la transmisión de un byte (8 bits) empleando esta configuración se realizará como muestra la figura 6:

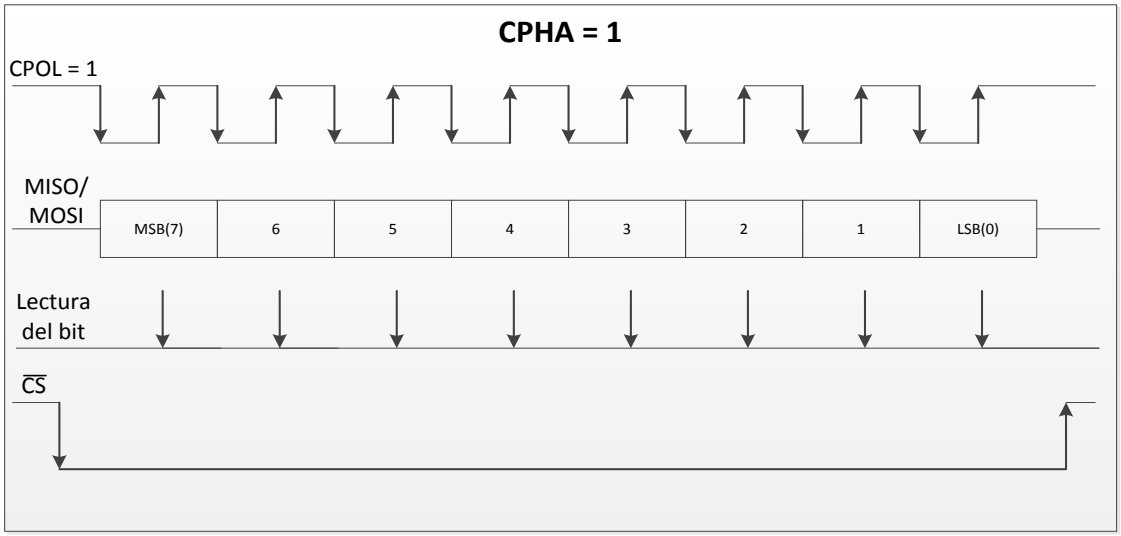


Figura 6: Modo 4 de transmisión SPI

2.2 Protocolo I2C

El protocolo I2C fue desarrollado por Philips Semiconductor en el año 1992 como un método para intercambiar información entre algunos de sus circuitos integrados. Este protocolo ha tenido diversas mejoras a lo largo del tiempo, donde se han ido incorporando nuevos modos de funcionamiento y mayores tasas de transferencia.

Se trata de un bus serie síncrono que emplea únicamente 2 líneas para las transmisiones entre los sistemas maestro y esclavos presentes en la comunicación. Dado que se trata de un bus serie, la transferencia de datos se realiza de forma secuencial, enviándose un único dato en cada periodo del reloj de sincronización.

2.2.1 Características

El bus I2C presenta las siguientes cualidades:

- Es un bus de dos hilos, emplea únicamente 2 líneas para realizar las transmisiones.
- La comunicación es half-duplex, es decir, solamente se puede enviar o recibir información y no ambas operaciones a la vez como en el SPI.
- Es un sistema maestro esclavo: Un dispositivo maestro se encarga de gestionar las operaciones de lectura y escritura de los esclavos.
- Se trata de un sistema multi-maestro: Permite que varios dispositivos maestros controlen el bus estableciendo un sistema de arbitración y detección de colisiones en caso de que varios quieran acceder de manera simultánea.
- Permite la conexión de numerosos dispositivos esclavos, marcando el número máximo de estos la capacidad del bus diseñado.
- Es un bus bidireccional, se emplea la misma línea de datos para realizar transmisiones maestro-esclavo y esclavo-maestro.
- Los esclavos tienen asignada una dirección única de modo que el direccionamiento se hace a través de la línea de datos.
- Las tramas se envían por bytes, es decir, la información se transmite en grupos de 8 bits.
- El tamaño de un mensaje no está limitado, pudiendo contener este varios bytes de información.
- Dispone de señal de seguimiento (*Acknowledge*) que permite al dispositivo transmisor (maestro o esclavo) identificar si el receptor ha recibido correctamente la trama.
- Posee diversos modos que establecen velocidades de transmisión desde los 100 Kbits/s hasta los 5 Mbits/s.
- Requiere un mínimo número de conexiones ya que únicamente emplea 2 líneas, lo cual hace que el circuito resultante de su implementación consuma pocos recursos.
- El direccionamiento integrado y el protocolo de transferencia de datos permiten que los circuitos integrados sean controlados por software.
- Existe un modo especial en el que la dirección del esclavo se compone de 10 bits, de modo que el número de esclavos que pueden ser conectados al bus aumenta significativamente (pasando de los 128 a los 1024).
- El sistema de arbitración que incluye es capaz de gestionar el acceso al bus de varios dispositivos mediante la detección del dispositivo que primero fija línea SDA a nivel bajo.

2.2.2 Señales del bus I2C

Este bus emplea 2 señales para realizar la comunicación entre un maestro y un esclavo como muestra la imagen expuesta a continuación:

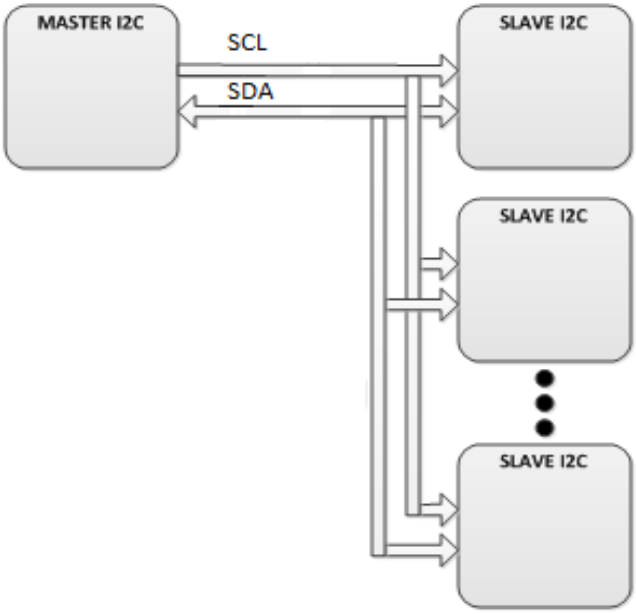


Figura 7: Conexión entre maestro y esclavos I2C

- **SCL**

Línea de reloj empleada para sincronización entre los dispositivos maestros y esclavos. Esta línea permite que la transmisión se realice en serie, enviándose o recibiéndose un dato en cada ciclo del reloj. La generación del reloj de transmisión es gestionada por los dispositivos maestros, de modo que permanecerá en nivel alto mientras no se realicen transmisiones. Dado que existe la posibilidad de que varios dispositivos maestros estén presentes en el bus y que los dispositivos esclavos pueden forzar la línea de reloj, es necesario que se disponga de un drenador abierto o un colector abierto para poder enviar y recibir la señal de reloj.

- **SDA**

Es la línea de datos empleada para las transmisiones maestro-esclavo y esclavo-maestro. A través de esta línea se envía toda la información, ya sea la dirección del esclavo, un byte de datos o una señal de asentimiento de recepción. Por tanto, es necesario que los dispositivos incluyan también en esta línea un drenador abierto o un colector abierto de manera que se puedan enviar y recibir datos a través de esta línea bidireccional. Mientras no se estén realizando transmisiones esta línea permanecerá a nivel alto.

2.2.3 Descripción del protocolo

Esta sección define el proceso para generar una transmisión entre un dispositivo maestro y un esclavo, tanto de lectura como de escritura, y las características que ésta puede presentar. En primer lugar se describirá el modo de funcionamiento de este protocolo y, posteriormente, se

detallarán cada de una de las características que permiten configurar la transmisión que se desea realizar.

- Funcionamiento

En el caso del bus I2C, el dispositivo maestro debe seguir unas pautas diferentes para enviar un dato al dispositivo esclavo o para recibirlo de este.

- ✓ Para iniciar cualquiera de las dos operaciones, el dispositivo maestro debe enviar una condición de inicio por el bus, indicando a los esclavos que una trasferencia va a comenzar.
- ✓ Seguidamente, el dispositivo maestro activará el reloj de sincronización y enviará por el bus la dirección del esclavo con el que desea comunicar y el tipo de operación a realizar, dentro de un mismo byte.
- ✓ El dispositivo esclavo seleccionado debe confirmar el direccionamiento enviando una señal de asentimiento (*Acknowledge*).
- ✓ En el siguiente ciclo de reloj comienza la transmisión, que dependerá del tipo de operación elegida:
    - Si la operación es de escritura el maestro enviará un byte de datos al esclavo, enviando un bit en cada ciclo del reloj SCL comenzando con el MSB del byte para terminar enviando el LSB. Finalizado el envío del byte, el esclavo enviará una señal de asentimiento en el siguiente ciclo para indicar que ha recibido el dato.
    - Si la operación es de lectura, será el esclavo quien envíe un byte de datos al maestro, de forma que cada bit se envíe durante un ciclo del reloj de sincronización empezando de nuevo por el MSB para terminar con el LSB. Cuando el byte ha sido enviado completamente, el maestro indicará al esclavo que ha recibido el dato mediante una señal de asentimiento.

  Estas operaciones se realizarán tantas veces como bytes contenga el mensaje a enviar o recibir.
- ✓ Cuando se ha enviado el ultimo byte del mensaje, el maestro actuará de forma diferente dependiendo del tipo de operación:
    - Si la operación es de escritura, el maestro esperará la señal de asentimiento del esclavo y posteriormente parará el reloj y enviará la condición de stop para indicar el final de transmisión o un reinicio para indicar el comienzo de una nueva transferencia.
    - Si la operación es de lectura, el maestro no enviará señal de asentimiento cuando se recia el último byte y parará el reloj de sincronización para posteriormente enviar una condición de stop y detener la transferencia o un reinicio para comenzar una nueva.

A modo de ejemplo, la siguiente imagen extraída del documento[REF-2] Especificación del bus I2C, muestra el desarrollo temporal de una transmisión empleando el protocolo I2C:

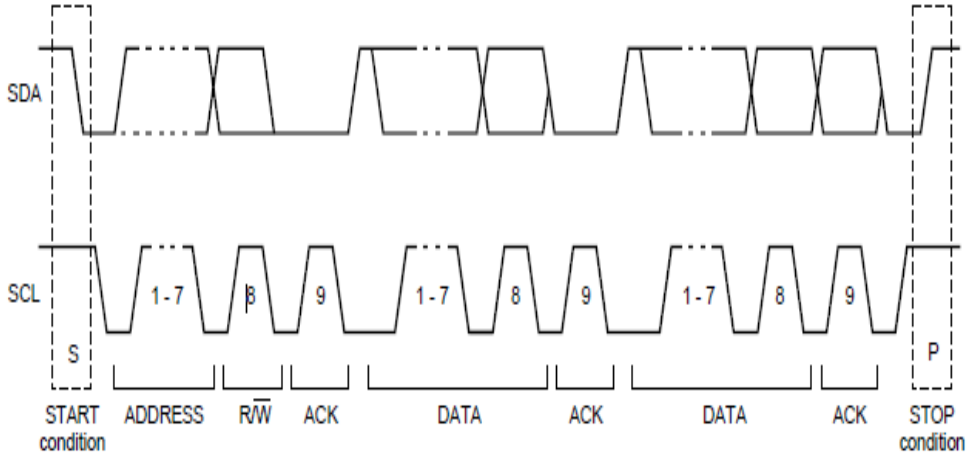


Figura 8: Ejemplo de transferencia I2C

- Modos

La especificación del protocolo I2C ([REF-2] Especificación del bus I2C) define diferentes modos de funcionamiento dependiendo de la velocidad establecida para las transmisiones y la posibilidad de enviar datos en ambos sentidos o en uno único:

  - Modos bidireccionales: Aquellos que permiten transmitir datos del dispositivo maestro al esclavo y viceversa. Existen los siguientes:
    - Modo Standard : velocidad de 100 Kbits/s
    - Modo rápido (*Fast mode*): velocidad de 400 Kbits/s
    - Modo rápido plus (*Fast mode plus*): velocidad de 1 Mbit/s
    - Modo de alta velocidad (*High speed mode*): velocidad de 3,4 Mbits/s
  - Modos unidireccionales: Aquellos que únicamente permiten realizar transmisiones desde el dispositivo maestro al esclavo. El esclavo no tiene capacidad de escribir en el bus, ni siquiera para enviar una señal de asentimiento. Estos modos se utilizan para aplicaciones de alta velocidad como pueden ser el control de pantallas LED de alta resolución. El protocolo I2c define un único modo unidireccional:
    - Modo ultrarrápido (*Ultra fast mode*): velocidad de 5 Mbits/s.

- Condiciones de Start, Stop y Repeated start

Dado que este bus no emplea líneas dedicadas para la selección de esclavo como el SPI y que únicamente presenta 2 hilos se definen una serie de condiciones para indicar a los esclavos el comienzo y final de una transmisión:

  - Condición de inicio (*start*):

  El comienzo de una transmisión se indica cambiando el nivel lógico de la línea SDA de alto a bajo mientras la línea de reloj SCL permanece en reposo, es decir, a nivel lógico alto. Resumiendo, se detectará el inicio de una transferencia cuando se produzca un flanco de bajada en SDA estando SCL a nivel alto como muestra la siguiente imagen extraída de la especificación del bus ([REF-2] Especificación del bus I2C):

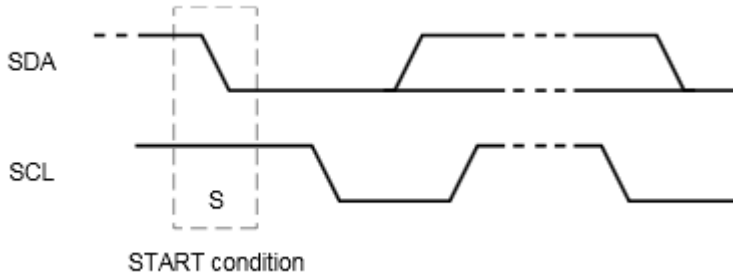


Figura 9: Condición de inicio de transmisión I2C

- Condición de parada (stop):

Una vez finalizada la transferencia de datos el dispositivo maestro indicará el final de ésta cambiando el estado de la línea SDA de nivel bajo a nivel alto mientras la línea de reloj SCL permanece en reposo a nivel alto. Es decir, se identificará el final de una transmisión cuando se detecte un flanco de subida en la línea SDA mientras la línea SCL está a nivel alto. La figura 10 extraída del documento de especificación del bus ([REF-2] Especificación del bus I2C) ejemplifica el fin de una transferencia:

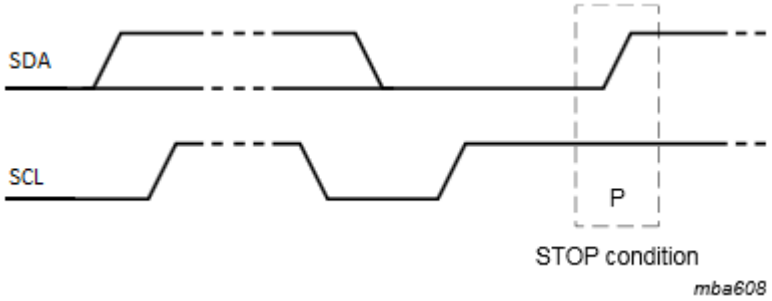


Figura 10: Condición de fin de transmisión I2C

- Condición de reinicio (*Repeated start*):

Esta condición se emplea cuando ha finalizado una transmisión y el dispositivo maestro que ocupa el bus desea continuar transmitiendo. Una vez transferido el ultimo byte del mensaje, el maestro enviará una condición de reinicio en vez de una de parada para direccionar a un nuevo esclavo. Esta condición se representa de manera análoga a la condición de inicio, es decir, cambiando el nivel de la línea SDA de alto a bajo mientras la línea de reloj SCL permanece a nivel alto como muestra la siguiente imagen:

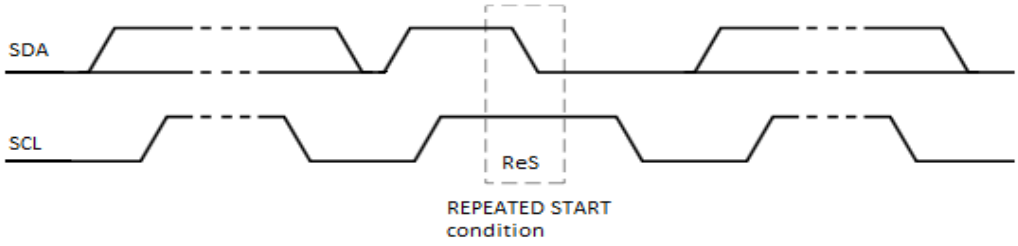


Figura 11: Condición de reinicio de transmisión I2C

Todas estas condiciones son gestionadas y generadas por el dispositivo maestro que desea realizar la comunicación y se definen de esta manera para asegurar que no hay problemas de interpretación entre las condiciones y los datos contenidos en el mensaje, tal como se aclara en el siguiente apartado

- Validez de datos

Los datos enviados han de ser leídos por el dispositivo receptor durante el tiempo en alto del reloj SCL por lo que deben ser escritos en el bus durante el tiempo que la línea SCL permanece a nivel bajo. De esta forma se asegura que el dato es estable durante el tiempo en alto del reloj. Esta especificación se relaciona con las condiciones de inicio y parada ya que permite diferenciar perfectamente si se está recibiendo un dato o una de las condiciones establecidas. El cambio en la línea SDA se producirá durante el tiempo a nivel bajo del reloj cuando se envían datos y durante el tiempo a nivel alto del reloj cuando se desea iniciar o finalizar una transmisión. La imagen presentada a continuación y obtenida de la propia especificación del bus ([REF-2] Especificación del bus I2C) indica los momentos en los que se permite realizar cambios en la línea de datos SDA con respecto a la línea SCL:

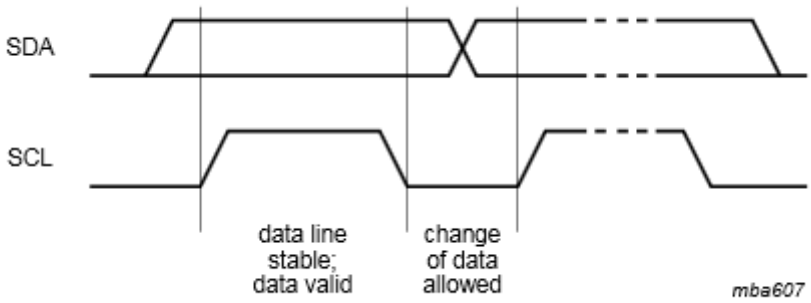


Figura 12: Validez de datos en transferencia I2C

- Formato de byte

Los datos son enviados en bytes, es decir, grupos de 8 bits. Cada transferencia comienza enviando el MSB del byte y finaliza con el envío de su LSB. La longitud de un mensaje no está restringida en número de bytes, es decir, cada mensaje puede contener un número distinto de bytes no existiendo un tamaño máximo.

- Dirección de esclavo y R/W

Después de enviar una condición de inicio, el dispositivo maestro envía por el bus un byte que contiene la dirección del esclavo con el que desea comunicar y el tipo de operación a realizar. La dirección del esclavo está compuesta de 7 bits (aunque existe un modo que permite la conexión de un número mayor de esclavos empleando 10 bits de dirección) que permiten identificar el esclavo a conectar. El direccionamiento del esclavo se realiza de manera univoca ya que cada esclavo posee una dirección única asignada distinta para cada uno. Existe la posibilidad de hacer una llamada general a todos los esclavos, empleada generalmente para configurar estos dispositivos. En caso de que los 7 bits de dirección sean 0, los esclavos identificaran la llamada como general y procederán a recibir los datos del maestro todos a la vez.

El último bit del primer byte enviado corresponde al tipo de operación que se desea realizar:

  - Operación de escritura: si el maestro desea escribir información en el esclavo enviará un cero lógico a continuación de los 7 bits de dirección.
  - Operación de lectura: si el maestro quiere leer información del esclavo mandará un uno lógico después de los 7 bit de dirección para que el esclavo comience a escribir la información en el bus.
- Señal de asentimiento (*Acknowledge*)

Una vez enviado un byte de dirección o datos el dispositivo transmisor fijará su salida de datos en alta impedancia para que el receptor pueda enviar una señal de asentimiento confirmando que el esclavo ha sido direccionado o que el byte de datos se ha recibido adecuadamente. La señal de asentimiento, generalmente conocida como *Acknowledge*) es generada por el dispositivo receptor, que puede ser el esclavo en operaciones de escritura o el maestro en operaciones de lectura. Esta señal se enviará durante el 9º ciclo del reloj SCL y consistirá en un cero lógico para indicar la correcta recepción o un uno lógico para indicar cualquier problema. Existen diversos motivos por los que puede no recibirse una señal de asentimiento:

- No hay receptor en el bus o la dirección enviada no corresponde a ningún esclavo.
- El receptor puede estar ocupado realizando otras operaciones y no puede comunicar.
- El receptor no entiende el dato enviado o lo considera erróneo.
- El maestro desea finalizar una comunicación. En este caso, el maestro no pondrá a nivel bajo la línea de datos SDA durante el tiempo a nivel bajo del 9º ciclo del envío, dejándola en nivel alto para poder enviar la condición de stop explicada previamente cuando el reloj vuelva a estar a nivel alto.

Para facilitar la comprensión de los conceptos implicados en la transmisión se puede observar la Figura 8 expuesta previamente, dónde se muestra el desarrollo completo de una transmisión del protocolo I2C.

3. Diseño e implementación

Una vez expuestos los fundamentos teóricos de los protocolos que se pueden analizar con el diseño presentado, este capítulo se encarga de describir la solución planteada. En primer lugar, se detallarán las características específicas del diseño mostrándose a continuación un diagrama de bloques del conjunto para facilitar su comprensión. Posteriormente, se detallará el diseño realizado para cada uno de los protocolos analizables describiéndose los bloques que lo componen y la funcionalidad que tiene cada uno de ellos.

3.1 Características específicas del diseño

El sistema diseñado posee las siguientes especificaciones:

- El diseño completo ha sido realizado empleando el lenguaje VHDL como herramienta para la generación del circuito resultante.
- Puesto que la implementación se ha realizado con lenguaje VHDL, muy utilizado para la implementación de los dispositivos maestros y esclavos de ambos buses, existen dos posibles usos. Cabe la posibilidad de implementar únicamente el diseño desarrollado en

una FPGA y conectarlo físicamente al bus. Como alternativa, se puede incluir la entidad de más alto nivel del diseño como un bloque más del proyecto del maestro o esclavo que se esté implementando de manera que el analizador quede contenido en la misma FPGA que el diseño a analizar.

- El diseño se ha dividido en diversos bloques con funcionalidades reducidas para facilitar su comprensión y permitir que los bloques se empleen en futuros diseños.
- Tanto la recepción de los mensajes como el manejo de estos se realiza dentro de la propia FPGA, enviando al ordenador el contenido de los mensajes con una estructura que facilite su posterior análisis.
- En el caso del protocolo SPI, el sistema está preparado para capturar mensajes con una longitud máxima de 128 bits por mensaje.
- En el caso del protocolo I2C, no existe una longitud máxima del mensaje a analizar. El sistema capturará cada byte del mensaje y lo mostrará por pantalla sin limitaciones de longitud.
- Las transmisiones entre la FPGA y el ordenador se realizarán a través de un puerto serie. Existe la posibilidad de que el ordenador no disponga de puerto serie (los modelos actuales no lo incorporan) por lo que será necesario emplear un adaptador de RS-232 a USB, puerto con el que cuentan todos los ordenadores actuales.
- El sistema es capaz de capturar tramas enviadas con el protocolo SPI abarcando el abanico completo de posibles velocidades. Podrán analizarse tramas enviadas con una velocidad de hasta 12,5 MHz, velocidad máxima disponible en el protocolo SPI diseñado por Motorola y consultable en la tabla 3-4 contenida en el documento [REF-1] Guía del bloque SPI de Motorola.
- Respecto al protocolo I2C, el diseño será capaz de capturar mensajes independientemente del modo seleccionado. Serán capturadas las tramas que se envíen a velocidades desde los 100 Kbits/s del modo Standard hasta los 5 Mbits/s del modo ultrarrápido.
- El envío de los datos a través del puerto serie se realizará mediante un módulo UART cuya velocidad de transmisión es configurable como se detallará más adelante. En nuestro caso se ha empleado una velocidad de 9600 baudios para dar robustez a la comunicación, aunque el circuito es configurable para funcionar a tasas más altas.
- Dado que es muy posible que la velocidad de captura sea mayor que la velocidad de transmisión por el puerto serie, el sistema incluye unas memorias FIFO capaces de almacenar 1024 mensajes de cada tipo. Con esto se consigue que los mensajes no se pierdan y se puedan enviar al ordenador cuando el módulo UART esté libre. Dependiendo de la capacidad en bloques RAM de la FPGA empleada este dato puede ser modificado para ampliar el tamaño de las memorias.
- Considerando la capacidad limitada de las memorias empleadas, existe la posibilidad de que, ante transmisiones muy rápidas con un contenido elevado de mensajes, se pueda perder algún mensaje de forma que sea capturado pero no enviado al ordenador. Para detectar este posible problema el sistema incluye un LED que se encenderá indicando que algún mensaje se ha perdido. El diseño dispone de un LED de mensajes perdidos para cada uno de los protocolos implementados.
- Se han contemplado posibles fallos en las transmisiones realizadas a través del protocolo SPI como son la longitud inadecuada del mensaje o la posibilidad de cortes durante la

transmisión de un mensaje. En el caso de que el mensaje posea una longitud inadecuada, el problema se identificará comprobando la longitud del mensaje transmitido al ordenador. Si se produce un corte en la transmisión, éste podrá ser identificado a través de los LED que incluye el diseño como se detallará posteriormente.

- También se han considerado los mismos posibles fallos en el caso del protocolo I2C de modo que, en caso de que un mensaje no contenga los bytes esperados, podrá detectarse a la hora de analizar los bytes recibidos en el ordenador. Si la transmisión se corta durante el envío de un byte, la identificación del problema dependerá del tipo de corte de transmisión. En caso de un corte de transmisión sin recepción de una condición de parada, el sistema mantendrá encendido un LED que indicará que se está capturando un mensaje. De éste modo, si el LED permanece encendido mucho tiempo podrá saberse que la transmisión se ha interrumpido sin finalizar. Si se detecta una condición de reinicio o parada, el sistema mandará al ordenador un byte que contendrá los bits recibidos hasta el corte y ceros en los bits que no se han recibido. Además, se indicará que el dispositivo receptor no ha enviado la señal de asentimiento. Ambas acciones permitirán identificar que el mensaje se ha cortado sin enviarse completamente.
- Además, el sistema incluye unos LED que indican que las memorias (tanto las reservadas para el SPI como para el I2C) están vacías, de modo que se pueda observar que todos los mensajes capturados han sido enviado ya a través del puerto serie.

3.2 Diagrama de bloques del sistema

A continuación se presenta un diagrama que muestra, a alto nivel, los elementos implicados y las conexiones existentes entre estos. En él puede observarse como el sistema desarrollado, el analizador de protocolos SPI e I2C, se conecta a las líneas que comunican los dispositivos esclavos con el maestro del bus. De manera análoga, el analizador se conectará a las líneas de reloj y datos del bus I2C, capturando la información enviada a través de estas. Por último, el diagrama incluye la conexión serie implementada entre el analizador y el ordenador dónde se realizará la visualización y análisis de los datos capturados.

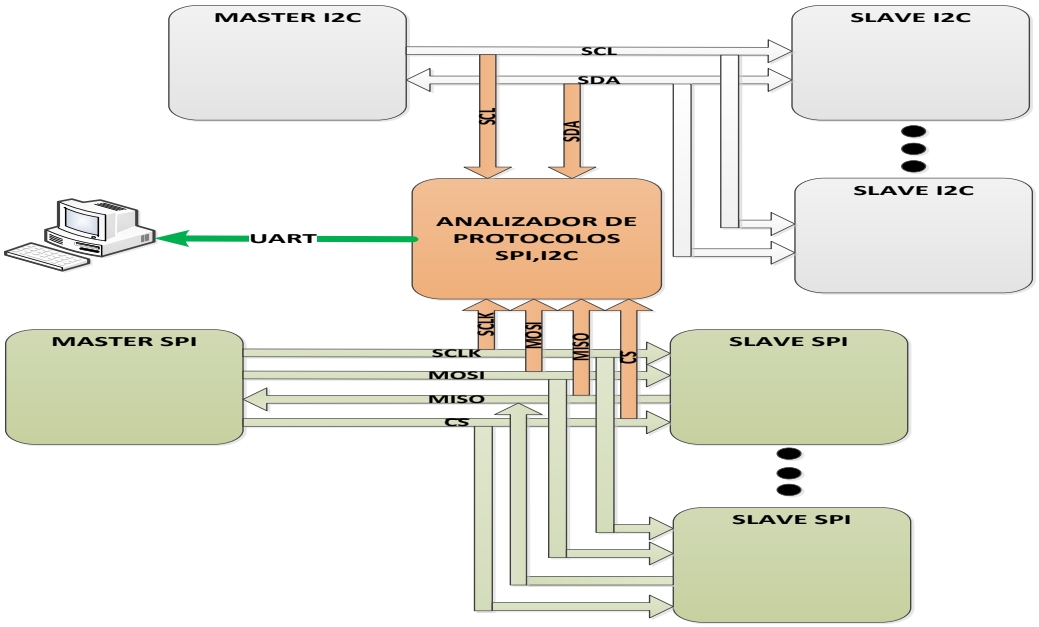


Figura 13: Diagrama de conexiones

3.3 Descripción de la solución

La siguiente sección describe en profundidad el diseño realizado. Cada uno de los bloques desarrollados será presentado y se describirá su interfaz con el resto de bloques y la funcionalidad que tiene. En primer lugar, se describirá el bloque de más alto nivel en la jerarquía, el cual aglutina el analizador de protocolos SPI y el de protocolos I2C. Posteriormente, se expondrán los bloques desarrollados para el análisis del protocolo SPI, empezando por el bloque de más alto nivel para finalizar con los más concretos. A continuación, se explicarán los bloques implicados en análisis del protocolo I2C de forma similar a la empleada con el protocolo SPI.

3.3.1 Analizador de protocolos

Este bloque contiene ambos analizadores, el analizador de protocolos SPI y el de protocolos I2C. Se trata del bloque de más alto nivel del diseño y su función es incluir todos los analizadores desarrollados para cada protocolo y gestionar cada uno de ellos.

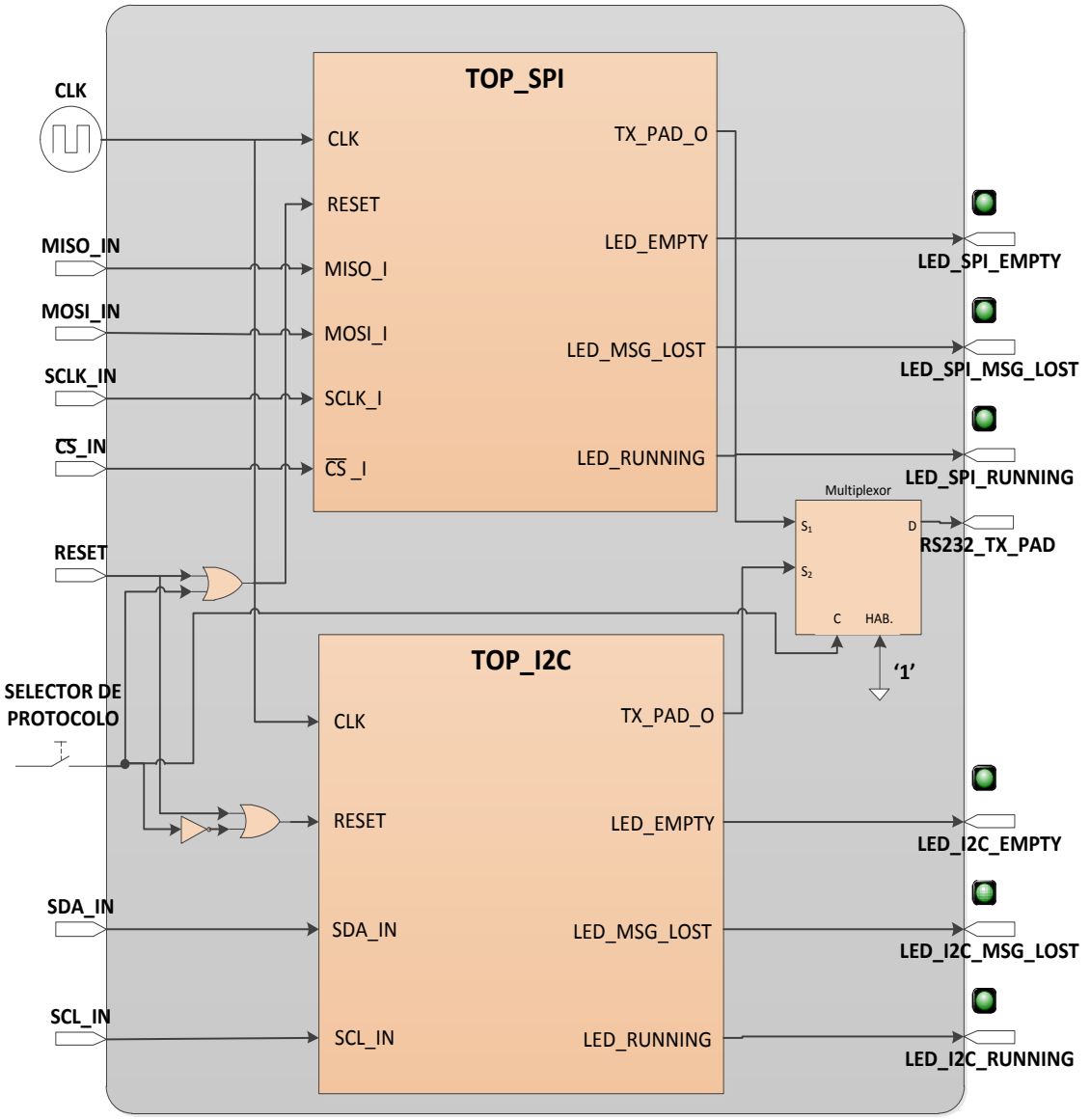


Figura 14: Diagrama de bloques del sistema completo

Interfaz del diseño (entradas/salidas del sistema)

La tabla 1 mostrada a continuación refleja el listado de puertos con los que cuenta el Analizador de protocolos:

Puerto	Tipo	Número de bits	Nivel activo	Valor de Reset	Descripción
CLK	Entrada	1	Flanco de subida		Entrada general de reloj del sistema
RESET	Entrada	1	'1'		Puerto para efectuar un reinicio del sistema
MISO_IN	Entrada	1	No aplica		Entrada de captura de datos transmitidos desde un esclavo al maestro del bus SPI que se está analizando
MOSI_IN	Entrada	1	No aplica		Puerto de entrada para la captura de los datos transmitidos desde el dispositivo maestro a cualquier esclavo del bus SPI
SCLK_IN	Entrada	1	No aplica		Entrada de análisis del reloj de sincronización del bus SPI
CS_IN	Entrada	1	'0'		Conexión de la línea de selección de esclavo del bus SPI
SDA_IN	Entrada	1	No aplica		Entrada para la captura de la línea de datos del bus I2C
SCL_IN	Entrada	1	No aplica		Puerto para la escucha del reloj de sincronización del bus I2C
RS232_TX_PAD	Salida	1		'1'	Puerto serie empleado para las transmisiones al ordenador
LED_SPI_EMPTY	Salida	1		'1'	Salida conectada a un LED que indica que el analizador SPI no tiene mensajes que enviar al ordenador

Puerto	Tipo	Número de bits	Nivel activo	Valor de Reset	Descripción
LED_SPI_MSG_LOST	Salida	1		'0'	Puerto de conexión a un LED para señalar que se ha perdido algún mensaje en el analizador SPI
LED_SPI_RUNNING	Salida	1		'0'	Salida a un LED que muestra cuando el analizador de protocolos SPI está capturando un mensaje
LED_I2C_EMPTY	Salida	1		'1'	Terminal preparado para un LED que marca que el analizador I2C no tiene mensajes capturados sin enviar al ordenador
LED_I2C_MSG_LOST	Salida	1		'0'	Conexión para LED que denota la pérdida de algún mensaje en el análisis de tramas I2C
LED_I2C_RUNNING	Salida	1		'0'	Salida conectada a un LED que señaliza que el analizador de protocolos I2C está capturando un mensaje.

Tabla 1: Interfaz del analizador de protocolos SPI e I2C

Funcionalidad

Este bloque se encarga de activar el analizador de un protocolo u otro y permitir que el analizador seleccionado envíe sus datos a través del puerto serie. Además, permite definir la interfaz del sistema completo ya que incluye todos los puertos necesarios para analizar ambos protocolos.

Mediante el interruptor de selección (entrada de selección de protocolo) se elige el protocolo que se desea analizar de modo que el analizador no activado queda en estado de reposo. Para ello, el sistema incluye una puerta OR conectada a la entrada de Reset de cada uno de los analizadores. De esta forma, cuando el Reset del sistema esté desactivado se evaluará la entrada de selección de protocolo. Si ésta tiene un nivel lógico de cero (el interruptor está en la posición de apagado) se activará únicamente el analizador de protocolos SPI ya que su puerta OR tendrá a su salida un cero, desactivando el Reset del analizador. Cuando el interruptor envíe un nivel lógico de uno, el analizador activo será el del protocolo I2C. Esto se consigue conectando el valor negado de la entrada de selección de protocolo a la puerta OR que hay en la entrada de Reset del analizador de protocolos I2C. De esta forma, la puerta OR tendrá a su salida un cero lógico (valor que desactiva el Reset) cuando el Reset general del

sistema no esté activado y el interruptor de selección de protocolo esté en la posición de encendido.

Por último, cabe destacar que este bloque se ha diseñado de esta manera para permitir que, en trabajos futuros, sea sencillo incluir un analizador para otro protocolo no siendo necesario modificar el diseño realizado para el resto de protocolos.

3.3.2 Analizador de protocolos SPI

Este subconjunto del sistema se encargará de capturas las tramas enviadas a través del bus de periféricos serie (SPI). En primer lugar se describirá la entidad de más alto nivel del diseño específico realizado para la captura de estas tramas describiéndose su interfaz con el resto del sistema y la funcionalidad que posee. A continuación, se describirán el resto de bloques diseñados e incluidos en el bloque de alto nivel, comenzando por el encargado de la captura de las tramas para continuar con el encargado de la gestión y envío al exterior de los mensajes y sus componentes.

3.3.2.1. TOP_SPI

Este bloque es la entidad de mayor nivel del analizador de protocolos SPI. Incluye la interfaz del analizador completo y contiene a los componentes encargados de la captura del mensaje (SPI_sampler) y su gestión y envío al ordenador (Control_SPI). Estos componentes y su conexión puede observarse en el diagrama de bloques expuesto a continuación:

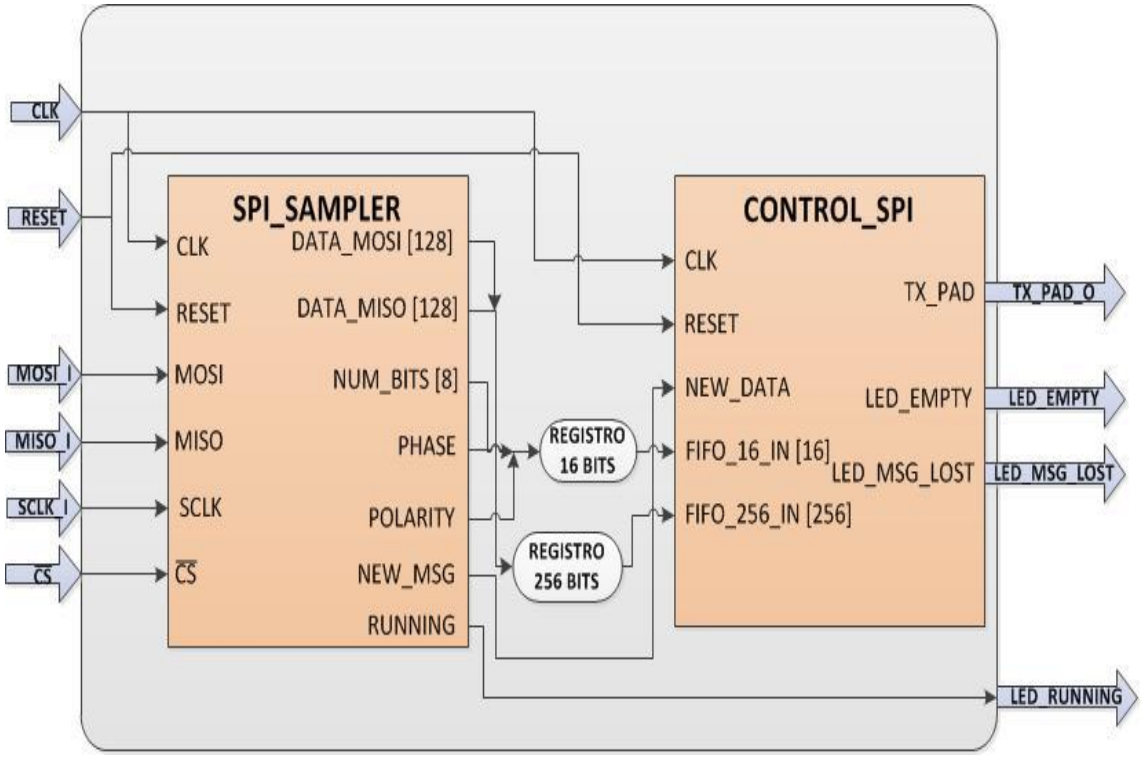


Figura 15: Analizador de protocolos SPI

Interfaz del diseño (entradas/salidas)

A continuación, la tabla 2 muestra el listado de puertos que representan la interfaz con el exterior del analizador de protocolos SPI:

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
CLK	Entrada	CLK	1	Flanco de subida		Entrada del reloj del sistema
RESET	Entrada	reset_SPI	1	‘1’		Puerto empleado para reiniciar el sistema
MISO_I	Entrada	MISO_IN	1	No aplica		Conexión a la línea MISO del bus SPI
MOSI_I	Entrada	MOSI_IN	1	No aplica		Conexión a la línea MOSI del protocolo SPI
SCLK_I	Entrada	SCLK_IN	1	No aplica		Puerto de recepción del reloj de SPI
CS_I	Entrada	CS_IN	1	‘0’		Conexión de la línea de selección de esclavo del bus SPI
LED_EMPTY	Salida	LED_SPI_EMPTY	1		‘1’	Puerto que indica al exterior que el analizador SPI no tiene mensajes por enviar
LED_MSG_LOST	Salida	LED_SPI_MSG_LOST	1		‘0’	Salida empleada para mostrar perdidas de mensajes en la captura

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
LED_RUNNING	Salida	LED_SPI_RUNNING	1		'0'	Terminal conectado a un LED que indica que el analizador está capturando un mensaje
TX_PAD_O	Salida	RS232_SPI	1		'1'	Puerto empleado para la transmisión serie al ordenador de los mensajes capturados.

Tabla 2: Interfaz del analizador SPI

Funcionalidad

El diseño de este bloque se debe a la necesidad de crear un sistema capaz de contener al analizador de mensajes y al sistema que los almacena y envía al ordenador. Además, se encarga de aglutinar los datos analizados para que se puedan almacenar en las memorias de manera adecuada como puede observarse en el diagrama de bloques mostrado anteriormente en la figura 15.

Los datos obtenidos (los cuales serán detallados en la siguiente sección) deben agruparse de una manera exacta para que se puedan almacenar en las memorias. Para ello, este sistema posee un registro de 256 bits de tamaño donde se agrupan los datos provenientes del dispositivo maestro y del esclavo. El tamaño de este registro proviene de la capacidad máxima de captura del analizador (128 bits por mensaje) y la configuración full-duplex del protocolo. Dado que la transmisión se realiza de manera simultánea en ambos sentidos (del maestro al esclavo y viceversa), el mensaje completo se compondrá de 256 bits considerando los datos transmitidos por el maestro (128 bits) y el esclavo (otros 128 bits).

Otro registro de 16 bits se encarga de agrupar los datos característicos del mensaje (longitud, polaridad del reloj y fase) para que éstos puedan ser también almacenados. Este registro obtiene los valores del sistema de captura y se rellena con ceros para facilitar la gestión posterior de los datos. Se ha realizado esta implementación por motivos de sencillez ya que esta opción facilitará la lectura de la memoria donde se almacenan y el envío de los datos que se detallarán en la descripción del bloque de control de los mensajes en la sección 3.3.2.3.

3.3.2.2. SPI Sampler

El bloque SPI_Sampler, denominado así por el término anglosajón Sampler (muestreador), es el encargado de efectuar la captura del mensaje que se envía a través de las líneas de datos del protocolo SPI. Esta sección describe la interfaz de este módulo así como su funcionalidad y el diseño que requiere para lograrla. La imagen mostrada a continuación describe el interfaz de puertos que incluye el sistema:

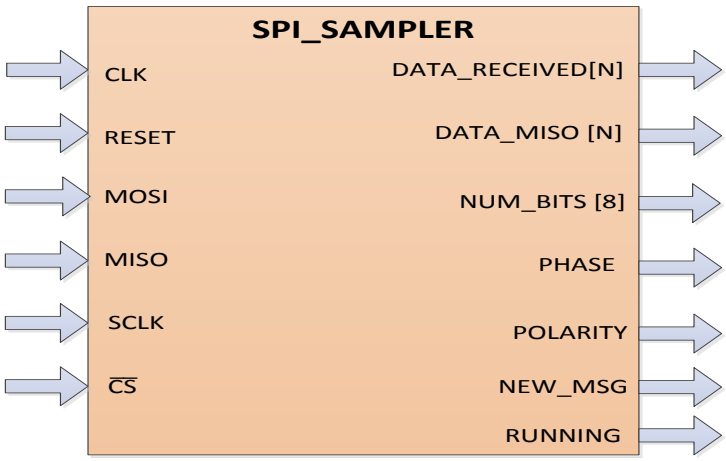


Figura 16: Interfaz del bloque SPI_Sampler

Interfaz del diseño (entradas/salidas)

- Genéricos:

Genérico	Tipo	Valor asignado	Descripción
G_MAX_NUMBER_OF_BITS	Natural	128	Parámetro empleado para definir el tamaño máximo del mensaje a capturar. Permite que se configure el número máximo de bits que contendrá un mensaje para poder adaptar el diseño a diferentes casos.

Tabla 3: Genéricos de la entidad SPI_Sampler

- Puertos

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
CLK	Entrada	CLK	1	Flanco de subida		Entrada del reloj empleada para sincronización del sistema
RESET	Entrada	RESET	1	‘1’		Terminal de reinicio del sistema

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
MISO	Entrada	MISO_I	1	No aplica		Conexión a la línea MISO del bus SPI
MOSI	Entrada	MOSI_I	1	No aplica		Conexión a la línea MOSI del bus SPI
SCLK	Entrada	SCLK_I	1	Flanco (de subida o bajada)		Puerto de recepción del reloj de sincronización del protocolo SPI
CS	Entrada	CS_I	1	‘0’		Conexión de la línea de selección del esclavo SPI.
DATA_MOSI	Salida	master_msg	128		Todos los bits a 0’	Vector de bits que contiene el mensaje enviado por el maestro SPI
DATA_MISO	Salida	slave_msg	128		Todos los bits a ‘0’	Vector de bits que contiene el mensaje enviado por el esclavo SPI
NUM_BITS	Salida	n_bits_msg	8		Todos los bits a ‘0’	Conjunto de bits que indica el número de bits que componen el mensaje SPI
PHASE	Salida	cpha_value	1		‘0’	Puerto empleado para transmitir la configuración de fase del mensaje capturado
POLARITY	Salida	cpol_value	1		‘0’	Valor de la polaridad del reloj del mensaje capturado
NEW_MSG	Salida	new_msg	1		‘0’	Puerto que señala que se ha capturado un nuevo mensaje

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
RUNNING	Salida	LED_RUNNING	1		'0'	Puerto que indica que el sistema está capturando un mensaje

Tabla 4: Interfaz del bloque SPI_Sampler

Funcionalidad:

Este bloque se encarga de analizar el estado de las líneas del protocolo SPI a las que se conecta para detectar si se realizan comunicaciones a través de ellas. Cuando se identifica el comienzo de una transmisión, el bloque captura los datos transmitidos entre el maestro y el esclavo seleccionado de bus, de manera que extrae los mensajes transferidos entre ellos, guardando como primer bit el MSB del mensaje y como último el LSB. Además, este bloque es capaz de identificar el modo de transmisión del mensaje reconociendo el valor de polaridad y fase que se ha configurado en la transmisión. También, dado que el tamaño de los mensajes no tiene por qué ser constante, este bloque se encarga de identificar el número de bits que componen cada mensaje capturado permitiendo además que se identifique que la longitud del mensaje enviado no tiene la longitud esperada. Por último, el sistema incluye una salida que refleja que se está capturando un mensaje. Esta salida permite identificar cortes en la transmisión ya que, permanecerá activa mientras no se identifique el fin de un mensaje, condición que no se cumplirá si la transmisión se ha interrumpido.

Implementación

Este diseño incluye numerosos componentes para conseguir obtener todos los datos necesarios:

- Un contador de flancos se encarga de calcular el número de bits que componen el mensaje. Cuando se detecta el inicio de un mensaje, el contador se activa y cuenta cada flanco del reloj SCLK que se recibe. Cuando finaliza la transmisión, el número de flancos detectados se envía a través de la salida NUM_BITS eliminando el LSB del vector, de modo que se obtiene el número de ciclos de SCLK recibidos (cada ciclo tiene 2 flancos, de subida y bajada) y por tanto el número de bits contenidos en el mensaje.
- Las entradas de datos MOSI y MISO poseen un registro que actualiza su valor cuando se detecta un flanco en el reloj SCLK para permitir la captura de cada bit del mensaje. Estos registros solo se actualizarán mientras se está realizando una captura gracias a una señal de habilitación controlada por la máquina de estados. Así se evita que se capturen datos cuando se produce un flanco en el reloj SCLK mientras no se están realizando transferencias, por ejemplo cuando se cambia la configuración de polaridad del reloj SCLK para enviar mensajes con un modo distinto al empleado anteriormente.
- Dos registros de desplazamiento se encargan de almacenar los bits que se han recibido, tanto de la línea MOSI como de la línea MISO. Estos registros están controlados mediante una señal de habilitación de recepción que permite que se almacene un nuevo bit y una

señal de fin de transmisión que vuelve a poner el registro al valor inicial para recibir un nuevo mensaje. Inicialmente, todos los bits de estos registros quedan fijados a 1, ya que éste es el valor por defecto de las líneas MOSI y MISO y la primera recepción se realizará al detectar un cero en la línea como se explicará posteriormente.

- La señal de habilitación de recepción se controla desde un proceso que activa dicha señal en diferentes condiciones:
    - Si se detecta un flanco de SCLK y la fase de la transmisión ya ha sido identificada, se habilitará la recepción cuando el contador de flancos sea igual al valor de fase identificado, es decir, se habilitará la recepción de datos en el flanco de reloj que no cambia la línea de datos para asegurar que se lee cuando el dato es estable.
    - Si la fase aún no se ha identificado pero se ha detectado el primer cambio de la línea de datos a 0, se habilitará la recepción en el flanco contrario al que se identificó el 0, de modo que se lee el siguiente bit cuando la línea de datos es estable.
    - Por último, cuando se detecte el primer cero en la línea de datos, se habilitará la señal de recepción para almacenar éste en el registro de desplazamiento.
- Otro proceso contiene las señales empleadas para detección de la fase del mensaje que permitirá capturar los datos en el momento adecuado. Dado que las líneas de transmisión MOSI y MISO están a nivel alto mientras el bus está en reposo, el sistema no puede identificar la fase con el nivel alto por lo que se emplea la identificación de un valor lógico de cero en la línea para identificar la fase en función de unas condiciones:
    - Si el primer cero se detecta en un flanco del reloj SCLK que no sea el primero, se puede identificar la fase del mensaje ya que sabremos en qué flancos está cambiándose el bit enviado y por tanto la fase del mensaje. Una vez identificado, el valor del LSB del contador de flancos se pasa negado a una señal para que los datos se lean en el flanco contrario al de cambio de bit del mensaje.
    - Por el contrario, si el primer cero se identifica en el primer flanco de SCLK, no podremos identificar aún la fase ya que no podemos saber con seguridad si ese bit se ha escrito en la línea antes del flanco (fase = 0) o después (fase = 1). Por ello, se espera a que se identifique un uno en la línea de datos, punto en el que si podremos saber cuál es la fase del mensaje de manera análoga al caso anterior.

  Para facilitar la comprensión de este proceso se recomienda revisar la explicación de los modos de transmisión SPI expuesta en el apartado 2.1.3.

- Por último, el sistema incluye una máquina de estados muy sencilla que se encarga de identificar el inicio y fin de las transmisiones y habilitar el resto de componentes mientras éstas se están realizando. La siguiente imagen muestra los estados de los que dispone esta máquina, los cuales serán descritos a continuación:

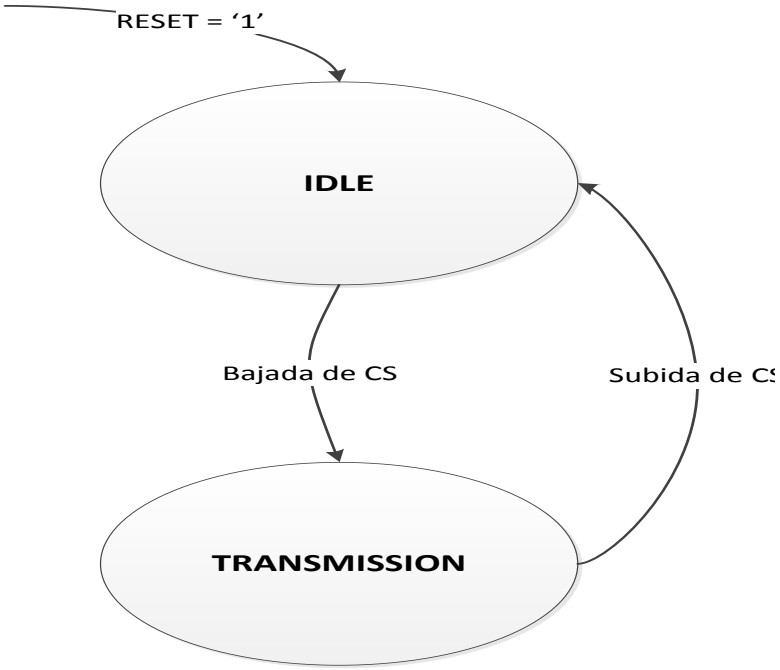


Figura 17: Máquina de estados del bloque SPI_Sampler

- La máquina se compone de los siguientes estados:
    - Idle: Estado de reposo del sistema. Cuando la máquina se encuentra en este estado la salida de RUNNING está a nivel bajo indicando que no se está capturando ningún mensaje. El contador de flancos es reseteado y el resto de componentes quedan inhabilitados para evitar problemas. Cuando se detecta una bajada en la entrada CS, condición que indica que el esclavo ha sido seleccionado para realizar una transmisión, se pasa al estado Transmission.
    - Transmission: Este estado marca que el sistema está capturando un mensaje por lo que la salida de RUNNING se pone a nivel alto indicando este hecho. Además, el resto de componentes son habilitados cuando se está en este estado, permitiendo que se realice la captura del mensaje. Cuando se identifica la subida de la entrada CS, condición que indica el fin de una transmisión, se activa la señal de fin de transmisión que hace que los datos capturados sean enviados a las salidas del sistema y se active la salida NEW_MSG indicando que se ha terminado de capturar un mensaje. En ese momento el sistema retornará al estado de reposo para esperar una nueva transmisión.

3.3.2.3. Bloque de control SPI

Este bloque tiene una importancia elevada en el diseño ya que permite que los datos sean almacenados y enviados de una forma adecuada al ordenador para su posterior análisis. En este apartado se describirá cual es la interfaz de dicho elemento, la cual puede observarse en la imagen presentada más abajo, describiéndose posteriormente las funcionalidad que tiene y los elementos implementados en su diseño.

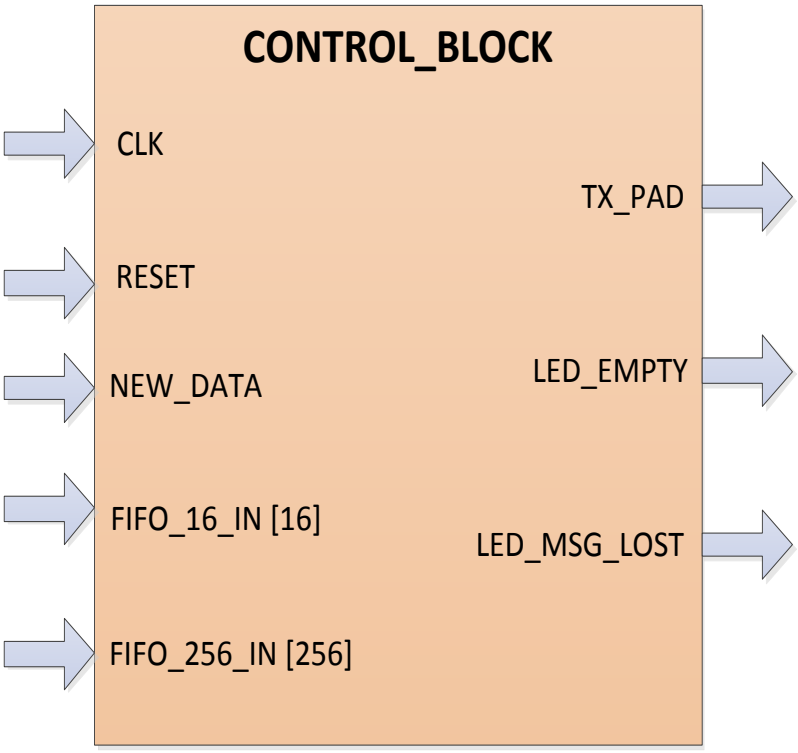


Figura 18: Interfaz del bloque de control SPI

Interfaz del diseño (entradas/salidas)

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
CLK	Entrada	CLK	1	Flanco de subida		Entrada del reloj general del sistema
RESET	Entrada	RESET	1	'1'		Puerto para reiniciar el componente
NEW_DATA	Entrada	new_msg	1	'1'		Entrada proveniente del bloque SPI_Sampler que indica que hay un nuevo mensaje que almacenar

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
FIFO_16_IN	Entrada	counter_and_ params	16	No aplica		Entrada de datos de la memoria FIFO de 16 bits
FIFO_256_IN	Entrada	master_slave_ data	256	No aplica		Entrada de datos de la memoria de 256 bits
TX_PAD	Salida	TX_PAD_O	1		‘1’	Salida del transmisor serie para efectuar los envíos de datos al ordenador
LED_EMPTY	Salida	LED_EMPTY	1		‘1’	Puerto que indica que las memorias FIFO están vacías
LED_MSG_LOST	Salida	LED_MSG_LOST	1		‘0’	Terminal que se conectará a un LED que indica que la pérdida de mensajes

Tabla 5: Interfaz del bloque de control SPI

Funcionalidad

Este bloque ha sido diseñado para almacenar los mensajes capturados por el SPI_Sampler y enviarlos de una forma adecuada al ordenador para su posterior análisis. Mediante la entrada NEW_DATA, el sistema identifica que se ha capturado un nuevo mensaje y habilita la escritura de los datos en 2 memorias FIFO. La primera memoria FIFO, con capacidad para 256 bits, almacena los 128 bits del mensaje enviado por el maestro (captura de la línea MOSI) y los del mensaje enviado por el esclavo seleccionado (captura de la línea MISO). La segunda memoria, con capacidad de 16 bits, almacena los 8 bits de longitud del mensaje, el valor de la polaridad del reloj SCLK en reposo (convertido a 4 bits mediante la adición de 3 ceros) y la fase del modo

empleado (de nuevo acompañada de 3 ceros para simplificar los envíos). En primer lugar, se enviará una cabecera seguida del identificar de mensaje y los datos que éste contiene.

Para reducir el tiempo de envío de mensaje y evitar que se envíen datos sin relevancia, el diseño únicamente envía los datos capturados, es decir, únicamente se enviarán los bits capturados y no el total de 128 bits que se almacenan por mensaje.

Los datos capturados se envían al ordenador en formato ASCII representándolos en este como caracteres hexadecimales, de modo que cada carácter hexadecimal representa 4 bits del contenido del mensaje. Para que esto sea posible se ha implementado un conversor que recibe 4 bits (un carácter hexadecimal) y devuelve el valor en ASCII de ese carácter.

Existe la posibilidad de que el mensaje contenga un número de bits que no sea múltiplo de cuatro, en cuyo caso el carácter incompleto será rellenado con ceros por la izquierda hasta alcanzar un valor múltiplo de 4 que se envíe al ordenador. Así, se consigue que la estructura del mensaje recibido no se vea afectada por la conversión de modo que el primer bit enviado tras los ceros será el MSB del mensaje y el último será el LSB. A modo de ejemplo, la siguiente tabla refleja la conversión que se realizaría en diferentes casos para el envío:

Nº de bits del mensaje	Mensaje	Mensaje con ceros adheridos	Equivalente hexadecimal
8	10101111	1010|1111	AF
9	110101111	0001|1010|1111	1AF
10	1010101111	0010|1010|1111	2AF
11	10110101111	0101|1010|1111	5AF

Tabla 6: Proceso de adición de ceros al mensaje SPI

La transmisión del mensaje comienza por los datos capturados del envío del maestro (captura de la línea MOSI) enviándose primero una pequeña cabecera que indica que los datos provienen del maestro y a continuación se envía el mensaje capturado comenzando con el MSB para finalizar con el LSB. A continuación se envía otra pequeña cabecera que indica que los datos posteriores corresponden a la captura del mensaje del esclavo (captura de la línea MISO) para pasar al envío del contenido del mensaje, de nuevo de MSB a LSB.

Finalizada la transferencia del contenido del mensaje, se enviará una cabecera seguida del número de bits contenidos en el mensaje, para lo cual es necesario disponer de un conversor de binario a BDC que será descrito posteriormente. Después, se enviará una cabecera seguida del valor de polaridad de SCLK con el que se transfirió el mensaje para acabar con otra cabecera que precede a la fase identificada en el mensaje.

Para finalizar, el sistema envía un carácter de final de línea y otro de retorno de carro haciendo que en el ordenador la representación salte de línea y pase al inicio de la siguiente para representar otro mensaje. De esta manera se consigue que en la pantalla del ordenador se represente un único mensaje en cada línea facilitando el análisis posterior de los datos recibidos.

Como ejemplo, un mensaje compuesto de 16 bits, considerando que el maestro y el esclavo han de enviar el mismo número de bits en la transmisión, sería representado de la siguiente manera en el ordenador:

MSG001: M-XXXX-S-XXXX-B016-PY-PHY

Figura 19: Representación general de un mensaje SPI

Donde X representa un carácter hexadecimal e Y el valor de la fase o la polaridad.

Si los datos enviados por el maestro fuesen "1111101000010100", el mensaje enviado por el esclavo fuese "1000101100101100" y se emplease el modo 2 de transmisión (CPHA=0 y CPOL=1) el mensaje mostrado en pantalla sería el siguiente:

MSG001: M-FA14-S-8B2C-B016-P1-PH0

Figura 20: Representación del ejemplo de mensaje SPI

Implementación

Considerando la complejidad del bloque expuesto y sus numerosas funcionalidades, se ha decidido que algunas de sus funciones sean realizadas por bloques específicos y que otras quedasen separadas en procesos dentro del código para simplificar la depuración de éste. La siguiente imagen recoge todos los elementos contenidos en el bloque así como algunas de las conexiones principales presentes entre ellos que facilitarán la comprensión del diseño y permitirán que posteriormente se detalle cada uno de los componentes implicados:

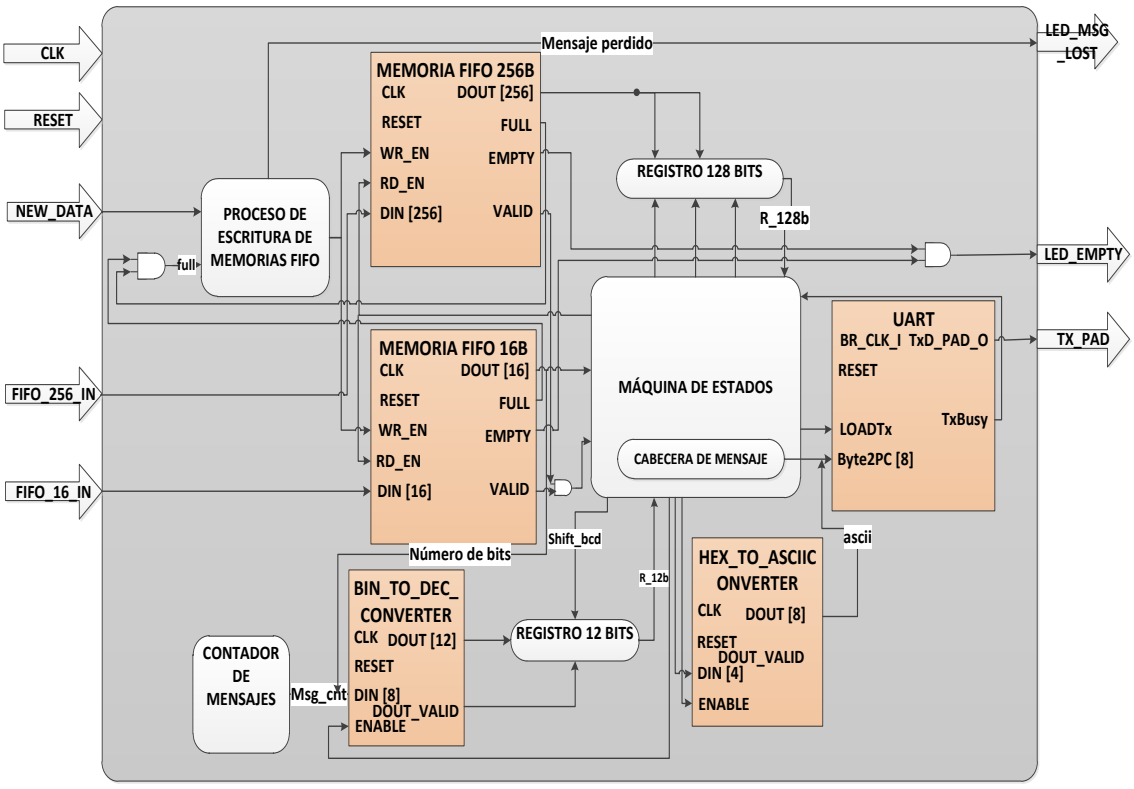


Figura 21: Diagrama de bloques del componente Control_SPI

Como puede observarse en la figura adyacente, este bloque incluye numerosos componentes en su desarrollo:

- Un proceso se encarga de realizar la escritura de los datos capturados en las memorias FIFO. Para ello, se detecta la activación de la entrada NEW_DATA y, si las memorias no están llenas, se habilita la escritura de las memorias para que almacenen los datos. Existe la posibilidad de que las memorias estén llenas cuando se detecta la recepción de un nuevo mensaje, caso en el que el sistema activará la salida LED_MSG_LOST, la cual permanecerá activa el resto del tiempo indicando que se ha perdido algún mensaje, aunque mensajes posteriores sí que hayan sido almacenados.
- Además, el bloque incluye un registro de 128 bits empleado para almacenar el contenido del mensaje del maestro o del esclavo. Dependiendo de unas señales controladas desde la máquina de estados, el registro puede copiar los 128 primeros bits de la salida de la FIFO de 256 bits (contendido del mensaje del maestro) o los últimos 128 bits (contenido del mensaje del esclavo. También se dispone de unas señales que permiten desplazar el registro 1,2 o 3 posiciones a la derecha y otra para realizar un desplazamiento de 4 posiciones a la izquierda haciendo posible la adición de ceros comentada anteriormente y permitiendo que se envíen los datos de manera secuencial conectando únicamente los 4 primeros bits de este registro al conversor de hexadecimal a ASCII comentará posteriormente.
- Este bloque incluye además un registro de 12 bits que almacena el resultado de la conversión de binario a BCD. Una señal controlada desde la máquina de estados permite desplazar 4 posiciones a la izquierda el registro para permitir que se conecten al conversor de hexadecimal a ASCII únicamente los 4 bits más significativos del registro.
- Se ha implementado también un contador de posiciones de cabecera que nos permite saber cuál será el siguiente carácter de la cabecera a enviar. El contador es controlado desde la máquina de estados mediante una señal que permite incrementar el valor y otra que se emplea para poner el valor a cero cuando finalice la transmisión de un mensaje.
- Otro contador es utilizado para calcular el número de bits del mensaje que han sido enviados ya. El contador tiene una señal que permite fijar el valor a 128 (número máximo de bits del mensaje del maestro o el esclavo) y otra entrada que permite restar 4 unidades al valor, de forma que cada vez que se envíe un carácter hexadecimal (4 bits) se reduzca el valor hasta llegar a cero, momento en el que el mensaje se habrá enviado completamente. Además este contador es utilizado para controlar los caracteres BCD enviados por lo que tiene una señal que fija el valor a 2 y otra que permite restar 1 unidad al contador para indicar cuantos caracteres BCD han sido enviados.
- El ultimo contador incluido en el diseño es un contador de 8 bits empleado para calcular el número de mensajes enviados. El valor máximo que alcanza es 255, longitud máxima habitual de las tramas de mensajes SPI. En caso de que se reciban más mensajes, el contador volverá a comenzar desde cero e incrementará su valor hasta llegar a los 255 de nuevo.
- Finalmente, el bloque basa su funcionamiento en una máquina de estados capaz de controlar la lectura de las memorias, el conversor de hexadecimal a ASCII, el conversor de binario a BCD y los contadores implementados. Dicha FSM queda representada en la siguiente imagen y sus estados se detallaran a continuación.

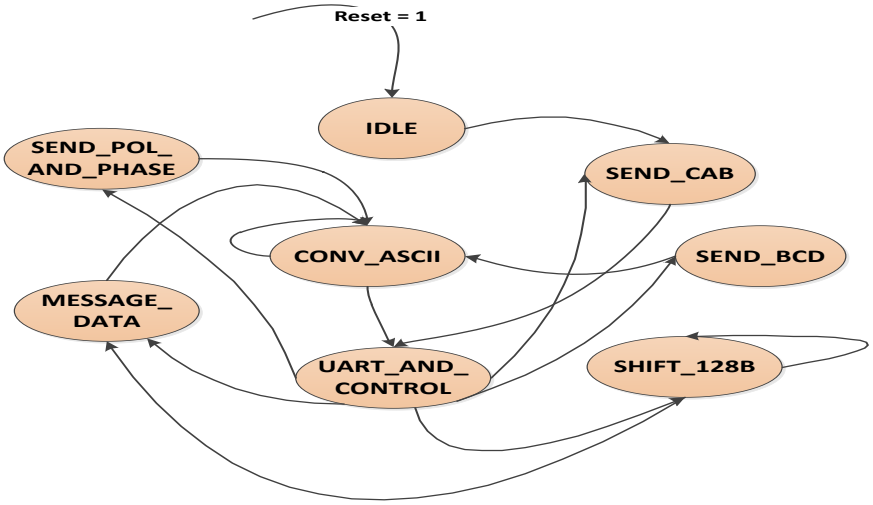


Figura 22: FSM del bloque Control_SPI

La máquina de estados arriba expuesta contiene los siguientes estados:

- Idle: Estado de reposo del sistema. Mientras las memorias no tengan ningún mensaje almacenado el sistema permanecerá en este estado. Cuando se detecte que las memorias contienen mensajes por enviar, el sistema pasará al estado send_cab
- Send_cab: Este estado se encarga de enviar los caracteres que componen la cabecera del mensaje mostrada anteriormente. Mediante el análisis del valor del contador de posiciones de cabecera se envía a la UART el valor ASCII (convertido a vector binario de 8 bits para adecuarlo a la entrada de la UART) correspondiente al carácter que se desea enviar (M, S, G…) y se pone a nivel alto la señal que activa la UART (load_tx) para que esta envíe el carácter por el puerto serie pasando al estado de uart_and_control.
- Send_bcd: en este estado se controla el envío de los datos que han sido convertidos a BCD, es decir, el número del mensaje que se está enviando y el número de bits que éste contiene. Evaluando el valor del contador auxiliar, el cual habrá sido previamente fijado a 2, se controla el envío de cada carácter BCD (centenas, decenas y unidades). Si el contador es mayor de cero, significa que las unidades todavía no han sido enviadas por lo que habrá que hacer la conversión a ASCII de las centenas o las decenas, pasando al estado conv_ASCII, y volver a este estado. Cuando el contador auxiliar valga cero indicará que el dato que se va a convertir son las unidades por lo que habrá que pasar al estado de conv_ASCII de nuevo pero a continuación se pasará al estado send_cab para mandar un nuevo carácter de cabecera. Para conseguir definir el estado de destino, una vez que se haya realizado la conversión a ASCII y se haya enviado el valor por el puerto serie, se emplea un vector de 4 bits denominado r_control que permite 16 combinaciones de transición que serán descritas en el estado uart_and_control donde se evalúa.
- Conv_ASCII: Este estado se encarga de manejar el conversor de hexadecimal a ASCII. Cuando la conversión ha sido completada, el conversor lo indica mediante una de sus salidas, el sistema pasa al estado uart_and_control y el resultado de la conversión será enviado a la UART para que transmita el valor ASCII correspondiente al carácter hexadecimal que se desea enviar.

- Shift_128b: Este estado se ha diseñado para realizar los desplazamientos necesarios en el vector de 128 bits que contiene el mensaje (del maestro o del esclavo) de forma que únicamente se transmitan los bits capturados en el mensaje. En este estado se evalúa la diferencia entre el contador auxiliar (el cual ha sido fijado con el valor 128 antes de pasar a este estado) y el número de bits que contiene el mensaje extraído de la memoria. Pueden darse varios casos:
    - La diferencia es mayor o igual a 4: En este caso, el registro de 128 bits es desplazado 4 posiciones a la izquierda y se reduce el valor del contador auxiliar en 4 unidades. Así se consigue eliminar del registro de 128 bits los bits que no corresponden al mensaje (cabe recordar que la memoria guarda 128 bits de la línea MOSI y otros 128 bits de la línea MISO independientemente del número de bits que realmente contiene el mensaje) y que el primer bit capturado se desplace hacia el MSB del registro ya que serán los 4 bits más significativos de éste los que se conecten al conversor de hexadecimal a ASCII para enviar los datos.
    - La diferencia es 3: En este caso, el bit más significativo del mensaje está ya dentro de las 4 primeras posiciones del vector de 128 bits pero el contenido del mensaje no es múltiplo de 4 (el bit más significativo del mensaje no puede agruparse con otros para componer un grupo de 4 bits), por lo que se añaden 3 ceros en la parte más alta del registro para permitir que se envíen caracteres hexadecimales como se ha explicado previamente (ver Tabla 6: Proceso de adición de ceros al mensaje SPI) y se pasa al estado message_data.
    - La diferencia es 2: este caso es similar al anterior pero en esta ocasión son los dos bits más significativos del mensaje los que se encuentran dentro de las 4 posiciones del registro de 128 bits que se envían al conversor. Por ello, se añaden 2 ceros en las primeras posiciones del registro y se realiza una transición al estado message_data.
    - La diferencia es 1: Caso de nuevo similar al anterior pero con los 3 bits más significativos del mensaje situados en las posiciones que se convierten. Ahora se añadirá únicamente un cero en la posición más alta del registro de 128 bits, haciendo que se componga un carácter hexadecimal con los 3 bits más significativos del mensaje y se pasa al estado message_data.
- Message_data: Cuando se entra en este estado el mensaje, sea el capturado de la línea MOSI o el de la línea MISO, ya está adecuado para que se envíen los caracteres hexadecimales. Los 4 bits más significativos del registro de 128 bits son enviados al conversor a ASCII y se realiza un desplazamiento de 4 posiciones en el registro para poder tener disponible el siguiente grupo de 4 bits (carácter hexadecimal) a convertir. A su vez, se activa la señal encargada de restar 4 unidades al contador auxiliar para controlar el número de bits que han sido enviados. Que el contador todavía sea mayor de 4 indica que aún quedan datos por enviar, por lo que se indicará mediante la señal r_control que debe retornarse a este estado para convertir otro grupo de 4 bits y enviarlo al ordenador. Por el contrario, si el contador tiene el valor de 0, se podrá saber que todos los datos han sido enviados y debe pasarse a otro estado.

- Send_pol_and_phase: Este estado se emplea para enviar al conversor a ASCII el grupo de 4 bits que contiene el valor de configuración de la polaridad o la fase del mensaje transmitido. Evaluando el valor de la señal r_control se envía la polaridad o la fase al conversor y se pasa al estado conv_ASCII para realizar la conversión.
- Uart_and_control: cuando la máquina pasa a este estado espera a que la UART pueda recibir otro dato para evaluar el valor de la señal r_control y pasar al estado adecuado de modo que se siga la secuencia correcta. La señal r_control toma diversos valores que especifican una transición y cada uno indica lo siguiente:

Valor de r_control	**Interpretación del valor y acciones realizadas**
0000	Se ha enviado un carácter de cabecera y debe enviarse otro más. Se incrementa el contador de posiciones de cabecera.
0001	Se ha enviado un carácter de cabecera y debe enviarse el primer digito BCD del número de mensaje. Se activa el conversor a BCD y, cuando la conversión ha finalizado, se pasa al estado send_bcd cargándose en el contador auxiliar el valor 2.
0010	Se ha enviado un carácter BCD y debe enviarse otro. Se desplaza el registro de 12 bits para poder convertir el siguiente dígito, se resta 1 al contador auxiliar y se pasa al estado send_bcd.
0011	Se ha enviado el dígito BCD de las unidades y debe enviarse la cabecera del mensaje del maestro. Se aumenta el contador de posiciones de cabecera y se pasa al estado send_cab.
0100	Se ha enviado la cabecera del mensaje del maestro y debe enviarse ya el mensaje. Se activa la lectura de las memorias FIFO y, cuando los datos a sus salidas son válidos, se carga el mensaje del maestro en el registro de 128 bits y se fija el contador auxiliar a 128. El siguiente estado será shift_128 para eliminar los bits leídos de la memoria que no pertenecen al mensaje capturado por el analizador.
0101	Se ha enviado un grupo de 4 bits del mensaje del maestro y debe enviarse otro. Se pasa al estado message_data para enviar más datos.
0110	Ha terminado de enviarse el mensaje del maestro (capturado de MOSI) y debe enviarse la cabecera del mensaje del esclavo (capturado de MISO). Se incrementa el contador de posiciones de cabecera y se pasa al estado send_cab para enviar el carácter adecuado.
0111	Se ha enviado la cabecera del mensaje del esclavo y éste debe ser transferido. Se carga en el registro de 128 bits el mensaje capturado de la línea MISO y de pasa al estado shift_128b para eliminar los datos que no corresponden al mensaje como se hizo con el mensaje del maestro.
1000	Se ha enviado un grupo de 4 bits del mensaje del esclavo y debe enviarse otro más, se pasa al estado message_data para gestionar el envío de éste

Valor de r_control	Interpretación del valor y acciones realizadas
1001	Ha terminado de enviarse el mensaje del esclavo y debe enviarse la cabecera del número de bits del mensaje. Se pasa al estado send_cab.
1010	Se ha enviado la cabecera del número de bits del mensaje y debe enviarse éste. Se activa el conversor a BCD y, cuando éste termina de convertir, se carga el valor 2 en el contador auxiliar y se pasa a send_bcd
1011	Ha terminado de enviarse la cabecera de la polaridad de SCLK y ésta debe ser transmitida. Se pasa al estado send_pol_and_phase
1100	Se ha enviado la polaridad del reloj y debe enviarse la cabecera que da paso a la fase. Se pasa al estado send_cab y se aumenta el contador de posiciones de cabecera para seleccionar el carácter adecuado.
1101	La cabecera de la fase ha sido transferida y su valor debe ser enviado. Se pasa al estado Se pasa al estado send_pol_and_phase.
1110	Todos los datos han sido enviados y debe cerrase el mensaje. En primer lugar se envía por el puerto serie el carácter ASCII de fin de línea y, posteriormente se envía el carácter correspondiente al retorno de carro para que siguiente mensaje comience en otra línea asignando a r_control el valor 1111 para que cuando acabe el envío el sistema vuelva al reposo.
1111	La transmisión del mensaje ha finalizado. Se incrementa el contador de mensajes enviados y se vuelve al estado de reposo.

Tabla 7: Interpretación de la señal r_control en el bloque control SPI

3.3.2.4. Memorias FIFO

El diseño realizado incluye 2 memorias FIFO empleadas para almacenar los mensajes y sus datos característicos, de forma que el primer mensaje almacenado sea el primer mensaje en leerse de la memoria. Esta sección se encarga de especificar la interfaz de estos módulos y concretar su funcionalidad e implementación. La figura mostrada a continuación refleja la interfaz que presentan estos módulos, donde N representa el ancho de la entrada y la salida de la memoria, parámetro que varía entre las dos utilizadas.

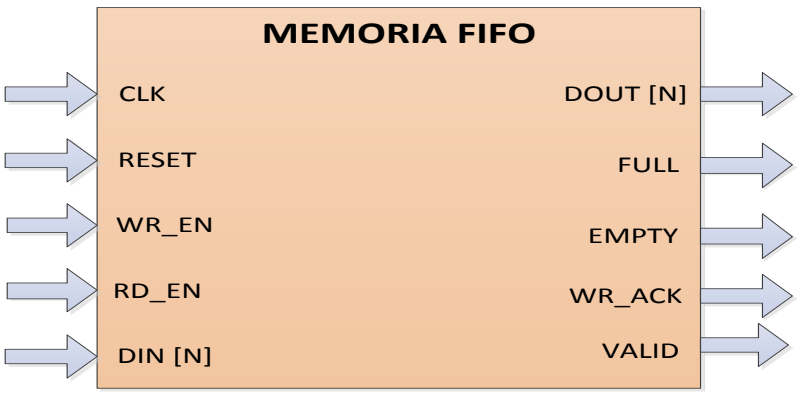


Figura 23: Interfaz de las memorias FIFO

Interfaz del diseño (entradas/salidas):

Considerando que se dispone de 2 memorias, la columna de señal a la que conecta refleja la señal conectada en ambas memorias y la columna de número de bits el tamaño del puerto en ambos caso. De esta manera se consigue simplificar la tabla mostrada a continuación:

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
CLK	Entrada	CLK	1	Flanco de subida		Entrada del reloj general del sistema
RST	Entrada	RESET	1	'1'		Puerto para reiniciar el componente
DIN	Entrada	FIFO_16_IN o FIFO_256_IN	16 o 256	No aplica		Entrada de datos de la memoria
WR_EN	Entrada	Wr_en	1	'1'		Puerto de habilitación de escritura
RD_EN	Entrada	Rd_en	1	'1'		Puerto de habilitación de lectura
DOUT	Salida	FIFO_dout_16 o FIFO_dout_256	16 o 256		Todos los bits a 0	Salida de datos de la memoria
FULL	Salida	full_16 o full_256	1		'0'	Salida que indica que la memoria está llena
WR_ACK	Salida	wr_ack_16 o wr_ack_256	1		'0'	Puerto que indica que la escritura ha sido correcta
EMPTY	Salida	empty_16 o empty_256			'1'	Salida que indica que la memoria está vacía, no contiene datos

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
VALID	Salida	valid_16 o valid_256			'0'	Puerto que indica que la salida de datos es válida y ésta puede ser leída.

Tabla 8: Interfaz de las memorias FIFO

Funcionalidad

Estas memorias se emplean para evitar que los mensajes capturados se pierdan ya que el envío por el puerto serie puede ser más lento que la captura, haciendo necesario que se almacenen los mensajes capturados. El sistema cuenta con dos memorias que se encargan de guardar diferentes datos:

- Memoria FIFO de 256 Bits: Esta memoria almacena los datos del mensaje capturados a través de las líneas MOSI y MISO. Dado que en cada una de las líneas se pueden capturar hasta 128 bits, esta memoria cuenta con un ancho de 256 bits, es decir, cada vez que se almacene información se guardarán 256 bits. En lo que se refiere a la profundidad, esta memoria es capaz de almacenar hasta 1024 mensajes de 256 bits.
- Memoria FIFO de 16 Bits: Este módulo es capaz de guardar 16 bits en cada una de sus posiciones. Se función es guardar los 8 bits que contienen la longitud del mensaje, la polaridad del reloj SCLK almacenada como 4 bits (000X) y el valor de fase que se ha configurado en la transmisión como otros 4 bits (000X). Los valores de polaridad y fase se han almacenado de esta manera para facilitar la lectura de la memoria y el manejo de los datos ya que para poder enviar el valor al ordenador, este debe ser convertido a un valor ASCII correspondiente a un carácter hexadecimal de 4 bits.

Implementación

Estos módulos no han sido directamente descritos empleando el lenguaje VHDL como los demás. En este caso se ha optado por emplear el *IP Core* (Módulo de propiedad intelectual) que facilita Xilinx para generar este tipo de memorias. Una vez seleccionado el módulo IP que se desea utilizar, el entorno de desarrollo lanza un asistente para definir los parámetros exactos del módulo. En nuestro caso se ha seleccionado la siguiente configuración, similar para ambas memorias exceptuando el ancho que varía como se especifica:

- Tipo de interfaz: Nativo
- Dominio de reloj de lectura/escritura: Reloj Común (CLK)
- Ancho de escritura: 16/256 (dependiendo de la memoria generada)
- Profundidad de escritura: 1024
- Manejo del puerto de escritura: se habilita el puerto de asentimiento de escritura (wr_ack) configurándose como activo a nivel alto.

- Manejo del puerto de lectura: se habilita el puerto de salida válida y se configura como activo a nivel alto
- Tipo de Reset: asíncrono
- Indicadores programables: no se configuran indicadores de casi llena o casi vacía.
- El resto de opciones quedan inhabilitadas dado que no serán necesarias.

Una vez configurados los parámetros del asistente el entorno de desarrollo se encarga de generar el archivo de síntesis resultante que se empleará para generar el archivo programable.

3.3.2.5. Conversor de Hexadecimal a ASCII

Este elemento ha sido diseñado como respuesta a la necesidad de adaptación de los datos capturados al formato ASCII que se emplea para comunicar mediante el puerto serie con el ordenador. Dado que los datos transmitidos son capturados en formato binario y su representación en formato ASCII supondría largas tramas de envío al ordenador (habría que enviar un carácter ASCII por cada uno de los bits transmitidos) se ha optado por agrupar los datos en grupos de 4 bits que compondrán un carácter hexadecimal y obtener el valor ASCII correspondiente a ese carácter. De este modo se reduce la cantidad de información a transmitir al ordenador y se facilita el análisis de los datos capturados.

A continuación, se definirá la interfaz que presenta este diseño y posteriormente se definirá su funcionalidad y la implementación que se ha realizado. La siguiente imagen refleja las conexiones que presenta el bloque y que serán definidas posteriormente:

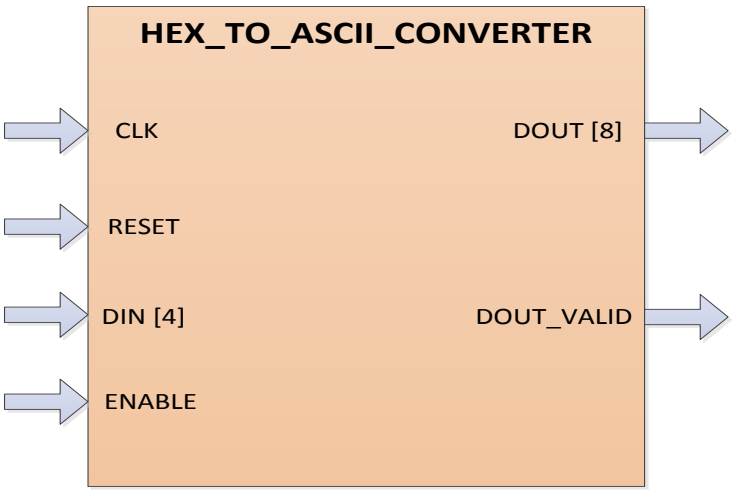


Figura 24: Interfaz del conversor de hexadecimal a ASCII

Interfaz del diseño (entradas/salidas):

La siguiente tabla define y especifica las características de cada uno de los puertos que presenta el conversor:

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
CLK	Entrada	CLK	1	Flanco de subida		Entrada del reloj común

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
RST	Entrada	RESET	1	'1'		Puerto para reiniciar el componente
DIN	Entrada	r_data2conv	4	No aplica		Entrada de datos a convertir
ENABLE	Entrada	enable_conv	1	'1'		Puerto para habilitar la conversión
DOUT	Salida	r_data_conv_o	8		Todos los bits a 0	Salida del valor convertido. Valor binario del carácter ASCII
DOUT_VALID	Salida	conv_valid	1		'0'	Salida que indica que la conversión ha finalizado

Tabla 9: Interfaz del conversor de hexadecimal a ASCII

Funcionalidad:

Como se ha comentado anteriormente, este componente se encarga de adaptar los datos del mensaje capturado para que se puedan transmitir adecuadamente al ordenador. Considerando el uso del puerto serie para dichas transmisiones, es necesario que los datos se envíen en formato ASCII para que puedan ser interpretados por el cliente serie empleado para la recepción. Por tanto, este bloque recibe un grupo de 4 bits que considera como un carácter hexadecimal y obtiene su equivalente decimal en ASCII. Concretamente, el bloque no devuelve el valor decimal en ASCII ya que la UART empleada no está diseñada para recibir valores decimales sino binarios. Por ello, el componente se encarga de obtener el equivalente decimal en ASCII del carácter hexadecimal recibido y convertir el valor a un vector binario de 8 bits que será lo que reciba la UART. A continuación, se muestra la tabla de valores empleados para conversión relacionando el grupo de 4 bits con su correspondiente valor hexadecimal, el equivalente ASCII de éste y el valor binario de dicho valor decimal:

Grupo de 4 bits	Valor hexadecimal	Equivalente ASCII decimal	Equivalente binario del valor decimal (8bits)
0000	0	48	00110000
0001	1	49	00110001

Grupo de 4 bits	Valor hexadecimal	Equivalente ASCII decimal	Equivalente binario del valor decimal (8bits)
0010	2	50	00110010
0011	3	51	00110011
0100	4	52	00110100
0101	5	53	00110101
0110	6	54	00110110
0111	7	55	00110111
1000	8	56	00111000
1001	9	57	00111001
1010	A	65	01000001
1011	B	66	01000010
1100	C	67	01000011
1101	D	68	01000100
1110	E	69	01000101
1111	F	70	01000110

Tabla 10: Valores de conversión de hexadecimal a ASCII

Nota: Para los valores alfabéticos del código hexadecimal se han empleado los equivalentes en mayúscula del código ASCII

Implementación:

Para conseguir la funcionalidad descrita anteriormente, este componente evalúa el estado de la entrada de habilitación (ENABLE). Cuando se detecta que está a nivel alto, el valor recibido en la entrada DIN es evaluado empleando un case para obtener el valor decimal correspondiente al carácter hexadecimal recibido. Finalmente, el valor decimal es convertido a un vector binario de 8 bits que se conecta a la salida DOUT.

Este método de implementación permite que los 8 bits correspondientes a cada uno de los valores decimales del código ASCII queden almacenados en una memoria ROM, de manera que el valor recibido en la entrada DIN actúa como dirección de lectura y reduce los recursos empleados por el diseño.

3.3.2.6. Conversor de binario a BCD

Como se ha expuesto previamente, el sistema diseñado incluye en la estructura del mensaje mostrado el número de mensaje mostrado y la longitud de este. Ambos valores son calculados en base binaria por lo que no pueden enviarse directamente, requieren que el valor sea convertido. Podría haberse empleado un sistema de conversión similar al presentado anteriormente pero en este caso, existen 256 posibles valores a convertir, por lo que el circuito

resultante consumiría numerosos recursos. Por ello, se ha optado por emplear otro método para hacer la conversión de una manera que requiera menos recursos. Concretamente, se ha empleado el algoritmo double dabble (traducido al español como recorre y suma 3) para conseguir que la conversión se realice de forma adecuada. De esta manera se consigue obtener el valor del contador a convertir como 3 dígitos BCD, formados cada uno de ellos por 4 bits, lo cual hace que sea muy sencillo realizar la conversión de cada uno de ellos para que se represente adecuadamente en el ordenador.

Tras esta introducción, se procederá a describir el interfaz que presenta este módulo así como la funcionalidad que desempeña y los recursos que se han empleado para su implementación. Para comenzar, la siguiente imagen refleja el conjunto de entradas y salidas del que dispone el módulo:

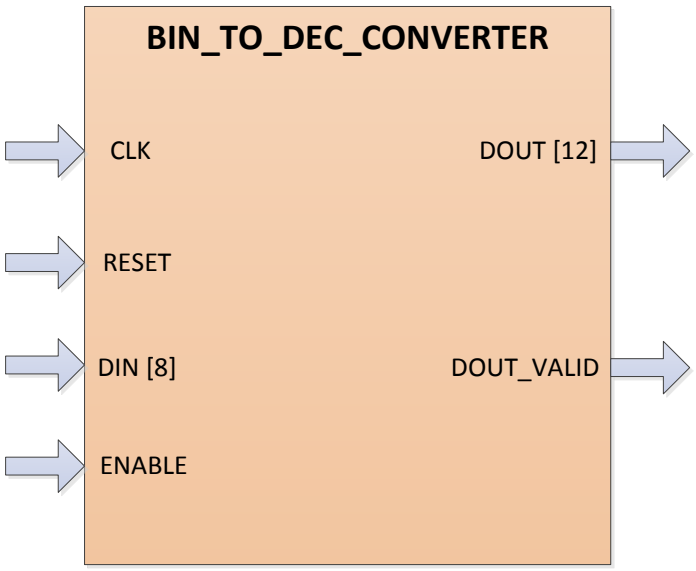


Figura 25: Interfaz del conversor de binario a BCD

Interfaz del diseño (entradas/salidas):

La tabla expuesta a continuación identifica cada uno de los puertos del componente y sus principales características

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
CLK	Entrada	CLK	1	Flanco de subida		Entrada del reloj de sistema
RST	Entrada	RESET	1	‘1’		Puerto para reiniciar el componente
DIN	Entrada	r_data2tx	8	No aplica		Entrada de datos a convertir

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
ENABLE	Entrada	enable_bcd	1	'1'		Puerto para habilitar la conversión
DOUT	Salida	r_bcd_o	12		Todos los bits a 0	Salida del valor convertido. Cada grupo de 4 bits representa un carácter BCD
DOUT_VALID	Salida	bcd_valid	1		'0'	Salida que indica que la conversión ha finalizado

Tabla 11: Interfaz del conversor de binario a BCD

Funcionalidad:

Este módulo ha sido diseñado para recibir valores binarios y devolver su equivalente en código BCD. El sistema es capaz de recibir 8 bits, los cuales pueden representar un valor máximo de 255, y devolver su codificación equivalente en BCD, representada mediante 3 dígitos BCD (centenas, decenas y unidades). Para conseguir este objetivo el sistema se basa en el algoritmo recorre y suma 3. Este algoritmo define un método para realizar conversiones de binario a BCD empleando desplazamientos de registro y una condición de suma. El proceso de conversión será el siguiente:

a. Desplazamos a la izquierda el registro de conversión introduciendo en el LSB un bit del número a convertir.
b. Si el valor de alguno de los caracteres BCD (grupo de 4 bits) es mayor de 4, sumamos 3 al valor del carácter BCD. Esto se realiza para evitar que el grupo de 4 bits se salga del rango definido de 0 a 9 para el carácter BCD.
c. Si todos los bits del número a convertir se han introducido en el registro de conversión, el valor ha sido convertido y se presenta como centenas, decenas y unidades.
d. Si todavía no se han desplazado todos los bits, se vuelve al paso 1 para repetir la operación.

Se suman 3 unidades al carácter BCD porque, acompañado de un desplazamiento a la izquierda posterior, equivale a sumar 6 unidades ya que desplazar a la izquierda supone multiplicar por 2 el valor. De esta manera es como si a cualquier valor superior a 4 se le sumasen 6 unidades, obteniendo un valor de 11 o superior dentro del carácter BCD. Al realizar el desplazamiento posterior a la adición el valor de la decena pasará al siguiente carácter BCD

haciendo que la transformación sea correcta. Para comprender mejor este concepto la siguiente tabla refleja la secuencia empleada para convertir el valor 12, expresado en binario como "1100", a su equivalente BCD:

Registro de conversión		Acción realizada
0000|0000	↓	Introducimos el primer bit del valor a convertir
0000|0001	↓	No es mayor de por lo que introducimos el siguiente bit
0000|0011	↓	No es mayor de 4 por lo que introducimos el tercer bit
0000|0110	↓	En esta ocasión sí es mayor de 4 por lo que sumamos 3 al digito BCD que estamos evaluando.
0000|1001	↓	Una vez obtenido el valor de la suma, introducimos el último bit a convertir
0001|0010		Todos los bits han sido introducidos, la conversión ha finalizado

Tabla 12: Ejemplo de conversión a BCD

Implementación:

Dado el funcionamiento secuencial del algoritmo recorre y suma 3, ha sido necesario implementar una máquina de estados que permita realizar los pasos en el orden adecuado. La FSM implementada incluye un contador de 3 bits que permite saber si los 8 bits del registro de entrada has sido desplazado al registro donde se realiza la conversión que es un registro de 12 bits ya que emplea 4 bits para representar las centenas, otros 4 para las decenas y 4 más para las unidades. Los estados definidos en la máquina comentada son los expuestos en la siguiente imagen:

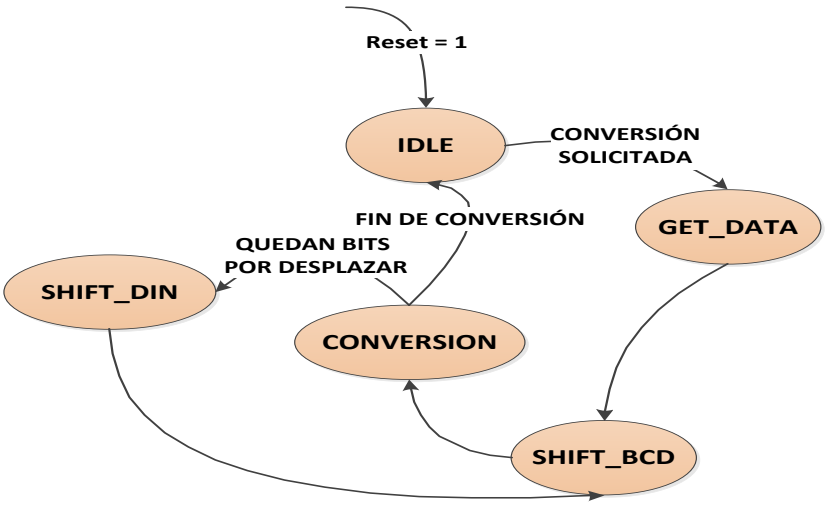


Figura 26: FSM del conversor de binario a BCD

Los estados empleados tienen la siguiente finalidad:

- Idle: Es el estado de reposo del sistema. Cuando se detecta un nivel alto se pasa al estado get_data.
- Get_data: Este estado se encarga de almacenar los 8 bits del valor que se desea convertir en un registro de desplazamiento. Cuando el valor ha sido almacenado se pasa al estado shift_bcd.
- Shift_bcd: Se introduce el MSB de registro de entrada en el registro empleado para la conversión. Se pasa al estado conversión.
- Conversión: Se evalúa el valor del contador para ver si se han realizado las conversiones necesarias. Si se han completado todos los desplazamientos, la conversión ha finalizado por lo que se activa la salida DOUT_VALID y se vuelve al estado idle. Si no, se realizada la comprobación en cada dígito BCD y se le suma 3 si es necesario para pasar al estado shift_din.
- Shift_din: En este estado se desplaza el registro de entrada para que el siguiente bit del dato de entrada quede en el MSB del registro. Esto se debe a que en el estado shift_bcd siempre se cogerá el MSB del registro de entrada para introducirlo al registro de conversión. Cuando el desplazamiento se ha realizado se pasa al estado shift_bcd.

3.3.2.7. UART

Este elemento se encarga de establecer la comunicación entre el diseño y el ordenador donde se visualizaran los datos. Dado que no es el tema central del proyecto, que se basa en la captura de los datos, se ha optado por emplear un módulo ya diseñado y catalogado como código abierto facilitado por el tutor del proyecto. El diseño empleado, es propiedad de Philippe Carton y puede obtenerse gratuitamente de la página http://opencores.org

A continuación, se muestra el interfaz que presenta este diseño así como se define su funcionalidad y los elementos que componen su implementación. La interfaz del módulo es la expuesta en la siguiente imagen:

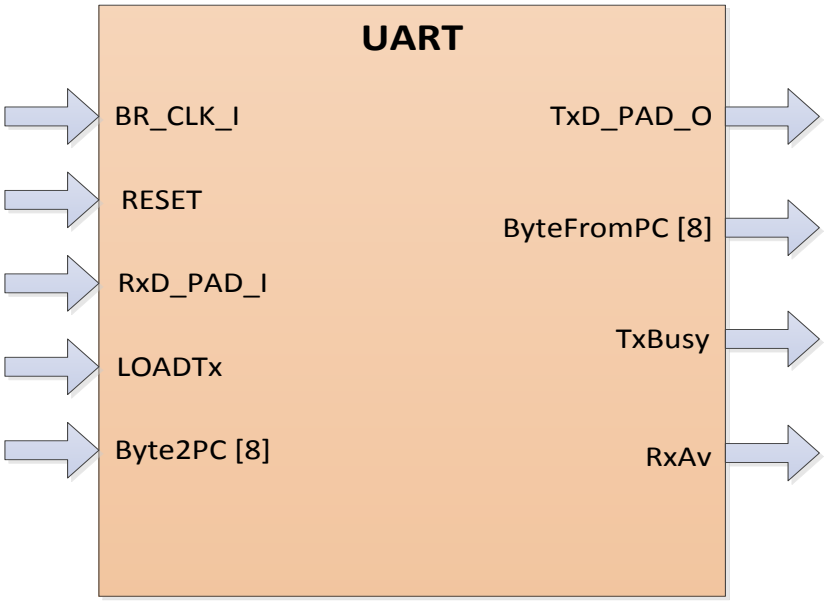


Figura 27: Interfaz del módulo UART

Interfaz del diseño (entradas/salidas):

Las tablas mostradas a continuación recogen los datos característicos del genérico que permite la configuración de la velocidad de transmisión del elemento y cada uno de los puertos del bloque UART:

- Genéricos:

Genérico	Tipo	Valor asignado	Descripción
BRDIVISOR	Entero	1302 (9600 baudios)	Parámetro empleado para definir la velocidad de transmisión que empleará el puerto serie. Pueden seleccionarse valores desde 0 hasta 65535 para obtener la velocidad deseada como se explicará posteriormente

Tabla 13: Genéricos de la entidad UART

- Puertos:

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
BR_Clk_I	Entrada	CLK	1	Flanco de subida		Entrada del reloj de sistema
RESET	Entrada	RESET	1	‘1’		Puerto para reiniciar el componente
Rx_PAD_I	Entrada	'1'	1	‘0’		Entrada de recepción serie RS232
LoadTx	Entrada	load_tx	1	‘1’		Puerto para habilitar una transmisión
Byte2PC	entrada	No conectado	8	No aplica		Vector que contiene el valor ASCII (en binario) que se desea enviar.
TxD_PAD_O	Salida	TX_PAD	1		‘1’	Salida de transmisión serie RS232

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
ByteFromPC	Salida	r_data2tx	8		Todos los bits a 0	Vector que contiene el valor ASCII recibido.
TxBusy	Salida	tx_busy	1		'0'	Salida que indica que el módulo está ocupado
RxAv	Salida	No conectado	1		'0'	Salida que indica que se ha recibido un nuevo byte

Tabla 14: Interfaz del módulo UART

Funcionalidad:

Este módulo está preparado para enviar y recibir datos a través del puerto serie. Permite la configuración de la velocidad de transferencia a través del genérico BRDIVISOR que se ha expuesto anteriormente. El valor del genérico se ajusta en función de la velocidad deseada según la siguiente fórmula:

$$\boldsymbol{BRDIVISOR} = {F_{BR\_CLK\_I}}/{Velocidad}_4 \quad \Leftrightarrow \quad \boldsymbol{Velocidad} = \frac{F_{BR\_CLK\_I}}{BRDIVISOR * 4}$$

Ecuación 2: Relación entre BRDIVISOR y la velocidad de transferencia

Dónde:

- BRDIVISOR: Parámetro genérico en el módulo. Debe emplearse un valor entero.
- $F_{BR\_CLK\_I}$: Frecuencia del reloj del sistema expresada en hercios (Hz).
- Velocidad: Velocidad de transferencia que se pretende alcanzar expresada en baudios (bits/s).

El diseño realizado en este proyecto envía datos al ordenador mediante el puerto serie pero está específicamente diseñado para que no reciba ningún dato de éste. Por ello, como puede observarse en la tabla de interfaz del módulo (ver Tabla 14) los puertos correspondientes al módulo de recepción no han sido conectados.

El bloque de control, definido previamente, se encarga de manejar este bloque asignando a su entrada de datos BYTE2PC el grupo de 8 bits correspondiente al carácter ASCII que se desea transmitir. Mediante la puesta a nivel alto de la entrada LOADTX, el diseño comienza la transmisión enviando un bit de inicio, fijado a nivel bajo, para posteriormente enviar el dato que tiene en su entrada BYTE2PC, comenzando por el LBS y terminando por el MSB. Cuando la

transmisión ha finalizado el diseño vuelve a poner la línea de datos a nivel alto, estado que mantiene para indicar el final de la transmisión y durante el reposo del sistema. La siguiente imagen ejemplifica la transmisión de 1 byte a través del puerto serie empleando el protocolo RS-232:

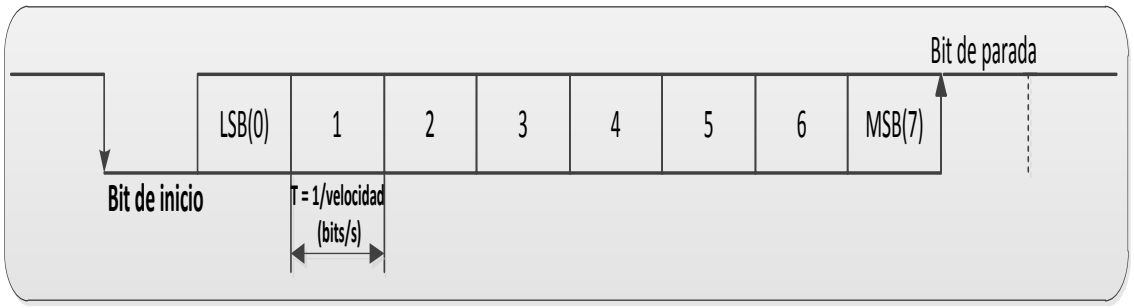


Figura 28: Ejemplo de transmisión RS-232

Implementación:

El sistema definido en este apartado se compone de números bloques que tienen una función específica. Estos bloques serán descritos a continuación indicándose su lista de puertos y su funcionalidad e implementación. La conexión de dicho bloques puede observarse en la siguiente imagen:

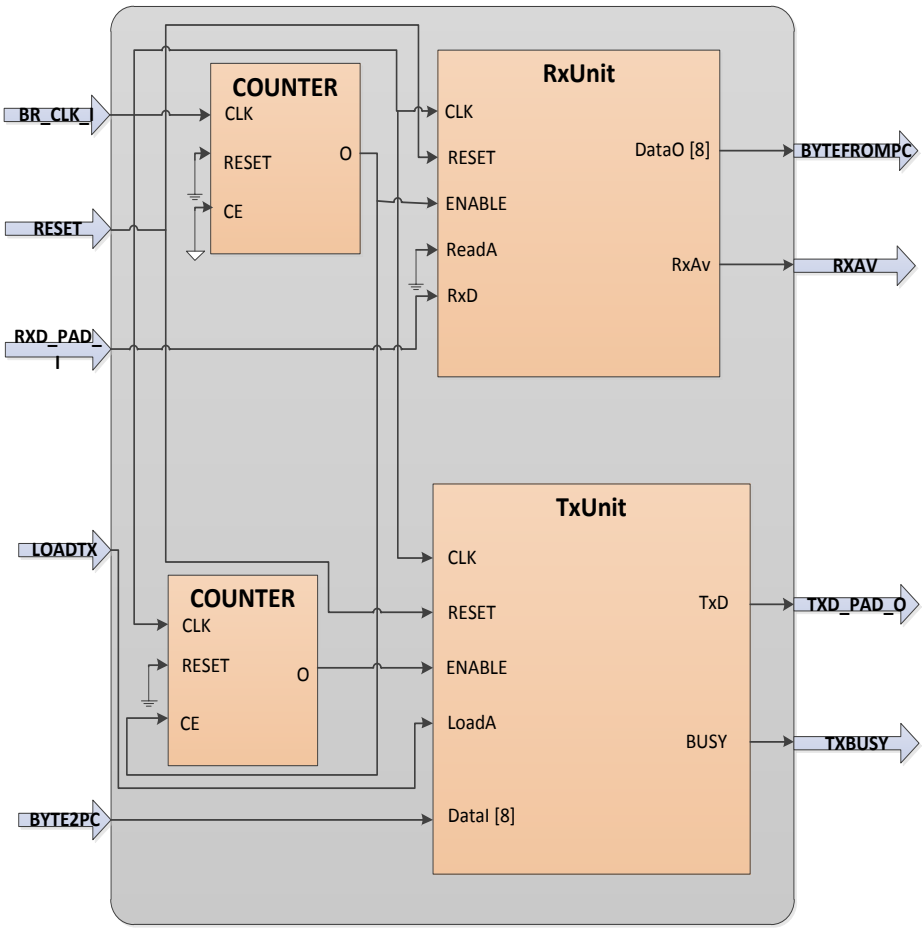


Figura 29: Diagrama de bloques UART

- Contador

Esta entidad compone un contador descendente de 16 bits. La interfaz del contador, que será descrita posteriormente junto con su funcionalidad y su implementación, aparece reflejada en la siguiente imagen:

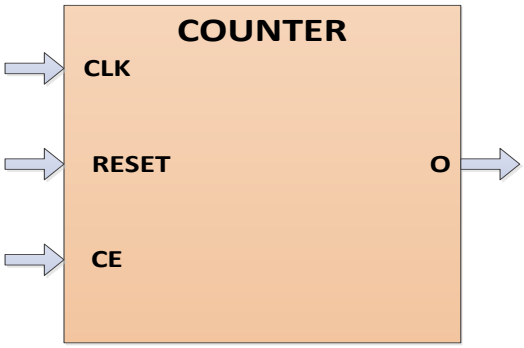


Figura 30: Interfaz del contador de la UART

Interfaz del diseño (entradas/salidas):

Las tablas mostradas a continuación recogen los datos característicos del genérico y cada uno de los puertos del contador implementado para la UART. Dado que, como puede observarse en el diagrama de bloques (Figura 29), se emplean dos contadores las tablas especifican los valores configurados para ambos:

- Genéricos:

Genérico	Tipo	Valor asignado	Descripción
Count	Entero	- Contador de recepción: 1302 - Contador de transmisión: 4	Parámetro empleado para definir el valor inicial de cuenta. Este parámetro está relacionado con la velocidad de transmisión.

Tabla 15: Genéricos del contador de la UART

- Puertos:

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
CLK	Entrada	BR_Clk_I	1	Flanco de subida		Entrada del reloj de sistema
RESET	Entrada	sig0	1	'1'		Puerto para reiniciar el componente

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
CE	Entrada	- Contador de recepción: sig1 Contador de transmisión: EnabRx	1	'1'		Entrada de habilitación
O	Salida	- Contador de recepción: EnabRx Contador de transmisión: EnabTx	1		'0'	Salida que indica que el valor de cuenta ha llegado a 0

Tabla 16: Interfaz del contador de la UART

Funcionalidad:

Los dos contadores empleados en este diseño se usan para gestionar la velocidad de transmisión que se desea utilizar. El contador empleado para la recepción se encarga de habilitar al bloque receptor para que muestree la línea de recepción de datos en el momento adecuado. Dicho contador está activo continuamente y emplea como valor inicial de su cuenta el valor fijado en el genérico de la UART, por lo que habilita el receptor 4 veces más rápido de lo que debería respecto a la velocidad de transmisión deseada. Este hecho se debe a que el receptor tomará 4 muestras de cada bit transmitido antes de almacenarlo de forma que se asegure que el bit guardado es el correcto.

En cambio, el contador de transmisión tiene fijado como valor máximo de cuenta 4 pero su señal de habilitación de cuenta está conectada a la salida del contador de recepción. De este modo, el contador de transmisión únicamente disminuirá su valor cada vez que el contador de recepción se ponga a cero. Esto permite que la transmisión de datos se realice a la velocidad deseada y da sentido a la ecuación de cálculo de velocidad de transmisión expuesta anteriormente (ver Ecuación 2: Relación entre BRDIVISOR y la velocidad de transferencia).

Implementación:

La implementación de estos bloques es bastante sencilla ya que emplean un proceso que contiene una variable encargada de almacenar el valor máximo de cuenta. Evaluando el estado de la señal de habilitación se resta una unidad al valor de cuenta o, si ésta está inhabilitada, se mantiene el valor anterior de la cuenta.

- Sincronizador

Este bloque está contenido dentro del transmisor de la UART. Su interfaz es muy sencilla y se muestra en la siguiente figura:

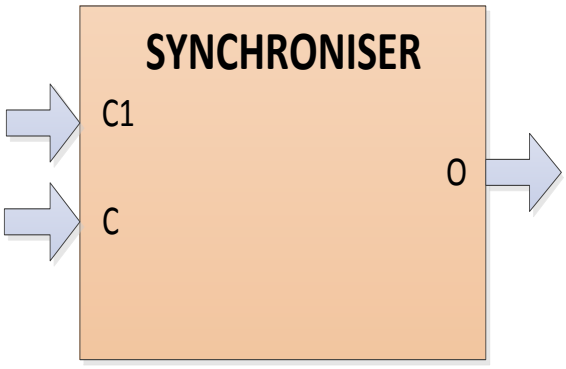


Figura 31: interfaz del bloque sincronizador

Seguidamente, se definirá la lista de puerto del componente y se describirán su funcionalidad e implementación:

Interfaz del diseño (entradas/salidas):

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
C	Entrada	clk	1	Flanco subida		Entrada del reloj
C1	Entrada	LoadA	1	'1'		Entrada asíncrona
O	Salida	LoadS	1		'0'	Salida síncrona

Tabla 17: Interfaz del contador de la UART

Funcionalidad:

La presencia de este bloque en el sistema se debe a la necesidad de adaptar las señales externas al dominio de reloj empleado por el resto de bloques. Dado que la UART puede ser controlada desde un dispositivo que no emplee el mismo dominio de reloj que el sistema, es muy posible que las entradas cambien su nivel de forma asíncrona. Para conseguir que todo el diseño sea síncrono se registra el valor de la señal entrante en cada ciclo de reloj y el diseño evalúa el valor de dicho registro. De este modo se consigue que la señal registrada esté sincronizada con el resto del diseño haciendo a éste completamente síncrono. Además, un segundo registro copia a su salida el valor del primero de modo que, gracias a la implementación realizada, la salida de este bloque fija su salida a nivel alto durante un único ciclo. De esta manera se consigue que el transmisor realice un único envío cada vez que su entrada LOADTX se pone a nivel alto, independientemente del tiempo que ésta permanezca así. En resumen, este bloque se encarga de sincronizar la entrada y crear un detector de flanco de ésta.

Implementación:

La implementación de este bloque es muy simple ya que únicamente se compone de un doble registro (el cual infiere dos biestables) que copian el valor de su entrada en cada flanco de

subida del reloj consiguiendo crear la sincronización necesaria. La salida síncrona del bloque está conectada a una puerta AND cuyas entradas son el primer registro y el valor negado del segundo. De este modo la salida se pondrá a nivel alto sólo cuando se produzca un flanco de subida en la entrada.

- Receptor

Este bloque no ha sido empleado en el proyecto ya que este no está diseñado para recibir datos a través del puerto serie, únicamente los transmite. No obstante, se detallará su interfaz y su funcionalidad para que pueda ser empleado en diseños futuros.

La imagen presentada a continuación, muestra el interfaz empleado por este diseño para comunicar con el resto de bloques:

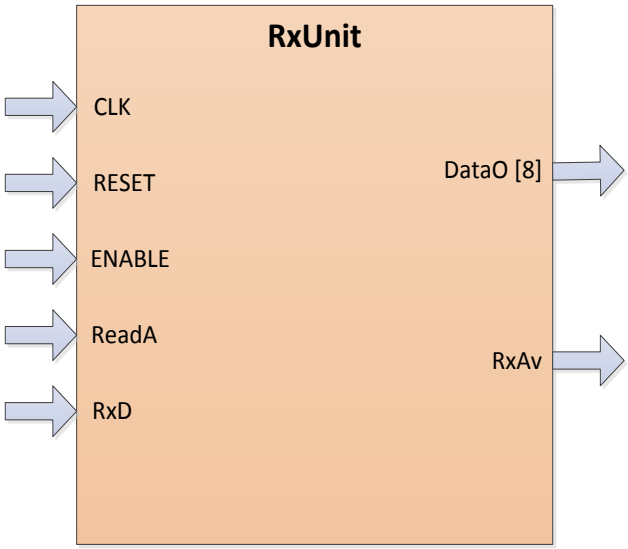


Figura 32: Interfaz del receptor de la UART

Interfaz del diseño (entradas/salidas):

La tabla expuesta a continuación identifica cada uno de los puertos del componente y sus principales características.

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
CLK	Entrada	BR_CLK_I	1	Flanco de subida		Entrada del reloj de sistema
RESET	Entrada	reset	1	'1'		Puerto para reiniciar el componente
ENABLE	Entrada	EnabRX	1	'1'		Puerto de habilitación de recepción de un bit

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
READA	Entrada	ReadA	1	'1'		Entrada para habilitar la lectura de un byte de la entrada serie
RXD	Entrada	RxD_PAD_I	1	'1'		Entrada serie de datos
RXAV	Salida	RxAvS	1		'0'	Salida que marca el fin de recepción de un byte
DATAO	Salida	RxData	8		Todos los bits a '0'	Byte recibido

Tabla 18: Interfaz del receptor de la UART

Funcionalidad:

Este bloque se encargaría de analizar la línea de recepción del puerto serie (RXD) y, cuando se le indique a través de su entrada READA, recibir el mensaje enviado a través de la línea mostrándolo en su salida DATAO e indicando que se ha recibido a través de su salida RXAV.

Implementación:

El diseño emplea un contador de 2 bits para realizar el muestreo de la línea serie a la velocidad requerida ya que el contador de recepción le indica que debe muestrearse 4 veces más rápido de lo necesario. Cada vez que el contador de recepción, expuesto anteriormente, indica que debe realizarse una muestra el receptor evalúa el valor de este contador y únicamente almacena el bit recibido cuando se realiza la segunda muestra. Así se asegura que la lectura se realiza en un momento en el que la línea de datos es estable y no cerca del cambio de bit transmitido.

Además, el diseño incluye otro contador, capaz de contar de 0 a 10, que se emplea para controlar el número de bits recibidos y finalizar la recepción. Este contador incrementa su valor cada vez que un bit es recibido y se vuelve a poner a cero cuando acaba la recepción para poder recibir otro mensaje.

Finalmente, el sistema está controlado mediante un case que evalúa el valor de dicho contador cada vez que se debe realizar una muestra, ejecutando diferentes acciones en función del valor de éste:

- Si el contador es igual a 0: Se evalúa la línea de datos buscando la identificación de un nivel bajo que indique que se ha recibido el bit de inicio de un mensaje. Si esta condición se cumple, el contador incrementa su valor. Si no es así, el sistema continuará esperando el bit de inicio.
- Si el contador es igual a 1: En este caso únicamente se incrementa de nuevo el contador cuando se muestrea la línea serie por cuarta vez. Este caso es extraño y se produce por un error en la implementación del diseño. Dicho error se produce al detectar el bit de inicio del mensaje ya que el contador de bits es fijado a 1 independientemente del valor del contador de muestras. Cuando se realiza la cuarta muestra dentro de la recepción del bit de inicio el contador de bits cambia su valor a 2 de modo que se incrementa dos veces durante la recepción del bit de inicio.
- Si el contador está entre 2 y 9: Se evalúa el contador de muestras realizadas para almacenar el dato recibido cuando se realice la segunda muestra como se ha indicado previamente. El dato será almacenado en un registro de 8 bits que posteriormente quedará reflejado en la salida DATAO mostrando el byte recibido. Cuando se realice la cuarta muestra de cada bit el contador de bits será incrementado.
- Si el contador es igual a 10: la recepción del byte ha finalizado por lo que se activa la salida RXAV para indicarlo y se copia el byte recibido a la salida DATAO. Los contadores son reiniciados para prepararse para recibir otro byte.

- Transmisor

Este otro componente sí que ha sido empleado en el diseño ya que el sistema debe transmitir al ordenador los datos capturados a través del protocolo SPI. Para ello, este bloque se encarga del manejo del puerto serie RS-232.

Esta sección define los puertos que incluye el transmisor de la UART así como su funcionalidad y características de implementación. La interfaz empleada por este elemento para comunicarse con el resto de componentes desarrollados puede observarse en la siguiente imagen:

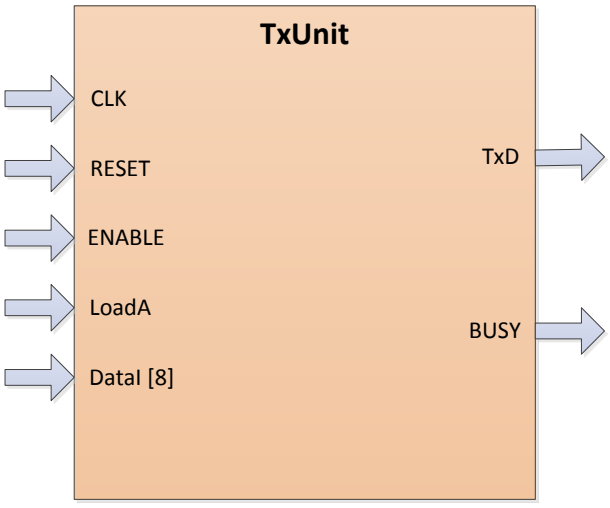


Figura 33: Interfaz del transmisor de la UART

Interfaz del diseño (entradas/salidas):

La tabla expuesta a continuación identifica cada uno de los puertos del componente mostrados en la imagen anterior

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
CLK	Entrada	BR_CLK_I	1	Flanco de subida		Entrada del reloj de sistema
RESET	Entrada	reset	1	'1'		Puerto para reiniciar el componente
ENABLE	Entrada	EnabTX	1	'1'		Puerto de habilitación de envío de un bit
LOADA	Entrada	LoadA	1	'1'		Entrada para habilitar la transmisión de un byte
DATAI	Entrada	TxData	8	'1'		Entrada que contiene el byte que se desea transmitir
BUSY	Salida	TxBusyS	1		'0'	Salida que marca que el transmisor está ocupado
TXD	Salida	TxD_PAD_O	1		'1'	Salida de transmisión del puerto serie

Tabla 19: interfaz del transmisor de la UART

Funcionalidad:

Este diseño es el encargado de realizar las transmisiones desde el sistema al ordenador a través del puerto serie disponible. Cuando se le solicita que realice una transmisión, este bloque envía el byte solicitado, que corresponde al carácter ASCII que se desea representar en el ordenador, a la velocidad que ha sido previamente establecida. El mensaje es enviado, tras

la transmisión del bit de inicio, comenzando por el LSB del vector y transmitiéndose un bit cada vez que lo indica el contador de transmisión hasta que se envía el MSB del vector. Mientras se está realizando una transmisión el sistema activa su salida BUSY para indicar que está ocupado y no puede recibir otro byte para enviar.

Implementación:

Este diseño cuenta con un contador con capacidad para contar de 0 a 10 que es empleado para controlar los bits del mensaje que han sido enviados. Además, el bloque incluye dos registros de 8 bits: uno empleado para almacenar desde la entrada el byte que se desea transmitir y otro, que toma el valor del anterior cuando comienza una transmisión, para controlar el bit que se está enviando. De esta manera se permite que el transmisor almacene un nuevo byte para transmitir mientras otro está siendo enviado. Así, se permite solicitar una nueva transmisión mientras un byte está siendo transmitido y que no haya que esperar a que la transferencia de éste finalice para poder requerir una nueva transmisión.

Por último, de manera análoga a la implementada en el receptor, se emplea un case que evalúa el valor de contador de bits cuando el contador del transmisor indica que debe enviarse un nuevo bit. Para realizar la transmisión de forma adecuada se usan una serie de condiciones:

- Si el valor de cuenta es 0: un nuevo mensaje debe comenzar por lo que el byte que se desea enviar es almacenado en el registro de entrada y el contador se incrementa para continuar con el envío.
- Si el valor de cuenta es 1: debe enviarse el bit de inicio de un mensaje por lo que la línea serie de transmisión se fija a nivel bajo y se incrementa el contador. El byte a enviar es almacenado en el segundo registro para que el primero pueda recibir un nuevo byte mientras el anterior se transmite.
- Si el contador está entre 2 y 9: Se enviará un nuevo bit a través de la línea serie (TXD) cuando el contador del transmisor, que previamente ha sido descrito, indique que debe realizarse el envío de un nuevo bit. El bit enviado será el correspondiente a la posición del registro que coincide con el valor del contador de bits. Ya que este contador es ascendente, el byte será enviado comenzando por su LSB para terminar con el MSB. El contador de bits será incrementado para continuar con la transmisión.
- Si el valor de cuenta 10: La transmisión habrá finalizado y el contador de bits se reiniciará para poder enviar otro byte si ha sido previamente solicitado.

3.3.3 Analizador protocolo I2C

Este subsistema comprende el diseño realizado para capturar los mensajes transmitidos a través del protocolo I2C y enviarlos al ordenador para su análisis posterior. El diseño se conectará a las líneas de reloj y datos del bus I2C para capturar la información transmitida y posteriormente la almacenará en unas memorias para transmitirlas a un ordenador empleando el protocolo RS-232 a través de un puerto serie.

A lo largo de esta sección se describirán los bloques diseñados y empleados para generar el circuito capaz de realizar la función indicada. En primer lugar, se describirán las características principales del bloque de más alto nivel de este diseño, detallándose los puertos que incluye,

su funcionalidad y la implementación que se ha realizado para obtenerla. Posteriormente, se describirán cada uno de los diseños contenidos dentro de dicho bloque, detallándose sus conexiones externas, su funcionalidad y la implementación que se ha desarrollado en ellos.

3.3.3.1. TOP_I2C

Este es el bloque de más alto nivel diseñado para el analizador de protocolos I2C. Contiene al bloque encargado de la captura de los datos y al diseñado para almacenarlos y enviarlos al ordenador con una estructura que facilite su comprensión.

A continuación, se presentará una imagen que refleja la interfaz que presenta este módulo para posteriormente describirse ésta en profundidad. Posteriormente se detallará su funcionalidad así como la implementación que contiene.

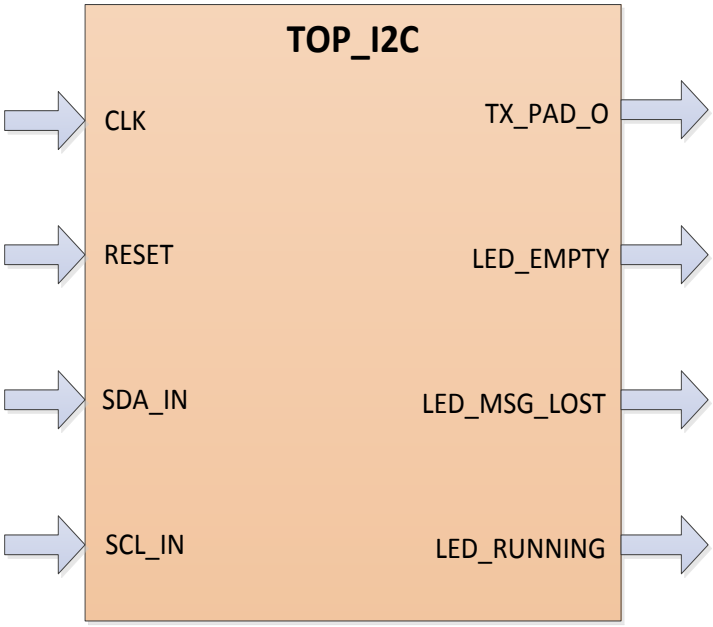


Figura 34: Interfaz del analizador de protocolos I2C

Interfaz del diseño (entradas/salidas)

A continuación, la tabla 20 muestra el listado de puertos que representan la interfaz con el exterior del analizador de protocolos I2C:

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
CLK	Entrada	CLK	1	Flanco de subida		Entrada del reloj del sistema
RESET	Entrada	reset_I2C	1	'1'		Puerto empleado para reiniciar el sistema

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
SDA_IN	Entrada	SDA_IN	1	No aplica		Conexión a la línea de datos serie del bus I2C
SCL_IN	Entrada	SCL_IN	1	No aplica		Puerto de recepción del reloj serie del bus I2C
LED_EMPTY	Salida	LED_I2C_ EMPTY	1		‘1’	Puerto que indica al exterior que el analizador SPI no tiene mensajes capturados para ser enviados
LED_MSG_LOST	Salida	LED_ I2C_ MSG_LOST	1		‘0’	Salida empleada para mostrar perdidas de mensajes en la captura
LED_RUNNING	Salida	LED_ I2C_ RUNNING	1		‘0’	Terminal conectado a un LED que indica que el analizador está capturando un mensaje
TX_PAD	Salida	RS232_ I2C	1		‘1’	Puerto empleado para la transmisión serie al ordenador de los mensajes capturados.

Tabla 20: Interfaz del analizador de protocolos I2C

Funcionalidad:

Este bloque ha sido diseñado para agrupar los elementos diseñados para la captura del mensaje I2C (I2C_Sampler) y el encargado de su almacenamiento y envío a través del puerto serie (Control_I2C). Además, este bloque permite adecuar las señales que conectan ambos bloques para que la recepción de la información sea correcta.

Implementación:

Este bloque tiene una implementación sencilla ya que solo se emplea para conectar los bloques mencionados anteriormente. Como peculiaridad, se encarga de agrupar el valor de la señal de asentimiento de un byte y la información relativa al final del mensaje para que ambas

puedan ser almacenadas en una memoria de 2 bits. La siguiente imagen refleja las conexiones realizadas entre los dos componentes instanciados:

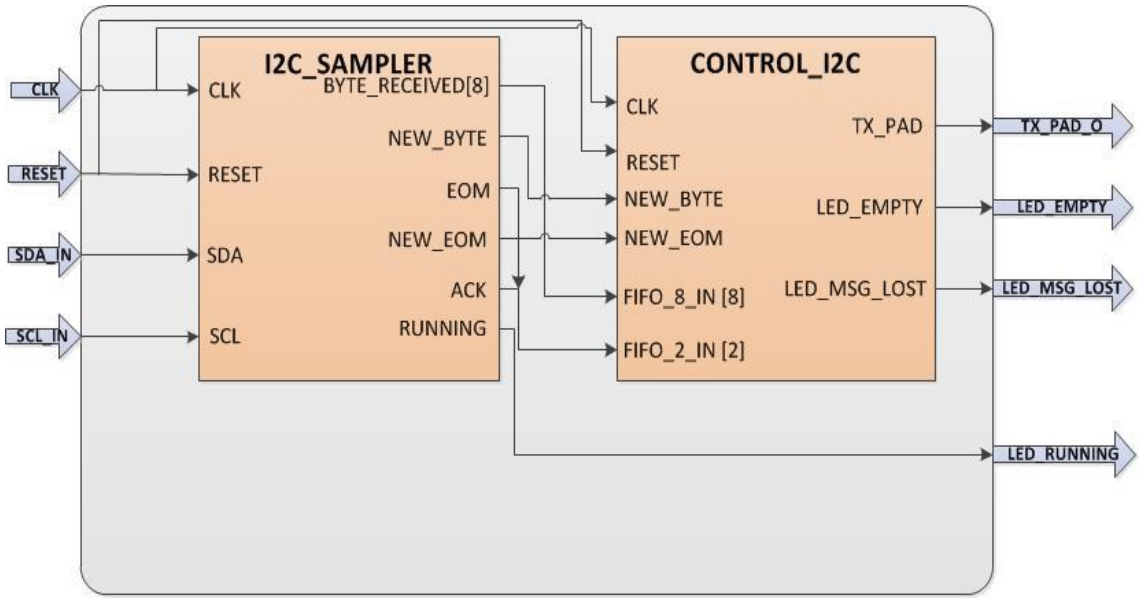


Figura 35: Conexiones entre bloques del analizador I2C

A continuación, se describirán los bloques empleados en el diseño de este analizador. Dado que algunos de los bloques usados son los mismos desarrollados para el analizador SPI, estos serán únicamente mencionados. Los bloques específicamente desarrollados para el analizador I2C serán descritos detalladamente, comenzando por el sistema de captura de datos para posteriormente describir el sistema de control de los mensajes.

3.3.3.2. I2C_Sampler

Este bloque es el encargado de realizar la captura de los mensajes transferidos a través del protocolo I2C e identificar fallos en la transmisión. Los puertos disponibles en este diseño se pueden observar en la imagen expuesta a continuación. Posteriormente esta interfaz será descrita junto con la funcionalidad detallada del diseño y la implementación realizada.

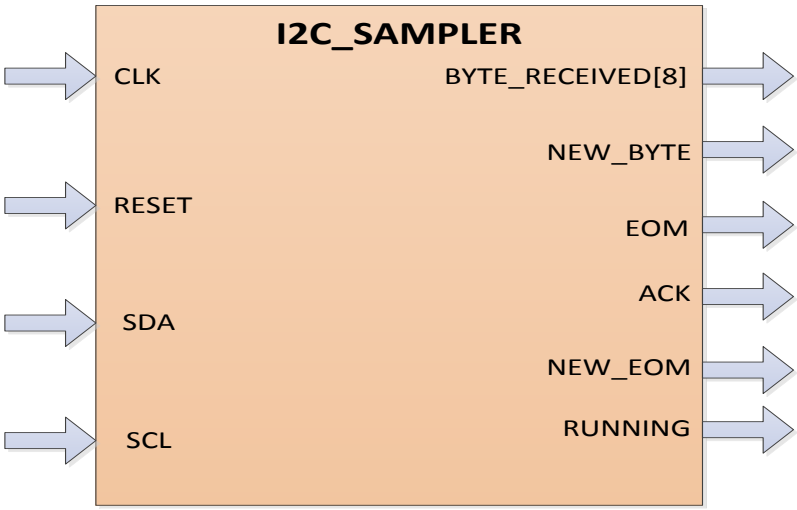


Figura 36: Interfaz del bloque I2C_Sampler

Interfaz del diseño (entradas/salidas):

La siguiente tabla recoge la lista de entradas y salidas disponibles en el componente I2C_sampler:

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
CLK	Entrada	CLK	1	Flanco de subida		Entrada del reloj del sistema
RESET	Entrada	reset_I2C	1	'1'		Puerto empleado para reiniciar el sistema
SDA	Entrada	SDA_IN	1	No aplica		Conexión a la línea de datos serie del bus I2C
SCL	Entrada	SCL_IN	1	No aplica		Puerto de recepción del reloj serie del bus I2C
BYTE_RECEIVED	Salida	byte_rec	8		Todos los bits a 0	Salida que contiene el byte capturado del protocolo I2C
NEW_BYTE	Salida	n_byte	1		'0'	Salida que indica que un byte ha sido capturado
EOM	Salida	s_eom	1		'0'	Salida que indica si el byte capturado era el último de un mensaje o no.
ACK	Salida	s_ack	1		'1'	Salida que indica el valor de la señal de asentimiento del byte capturado

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
NEW_EOM	Salida	n_eom	1		'0'	Puerto para indicar que se ha detectado si el byte anterior era el final de un mensaje
RUNNING	Salida	LED_RUNNING	1		'0'	Salida que indica que el sistema está capturando un mensaje

Tabla 21: Interfaz del bloque I2C_Sampler

Funcionalidad:

Este bloque ha sido diseñado para capturar la información que se transmite a través del bus I2C e identificar posibles fallos en la transmisión. El sistema identifica la recepción de una condición de inicio de mensaje para comenzar la captura de cada byte. Tras esto, se captura cada uno de los bits contenidos en el byte cuando el dato es estable, es decir, durante el tiempo en nivel alto del reloj de sincronización (SCL). Cuando se ha recibido el byte completo, este componente activa su salida NEW_BYTE para que el byte se almacene y captura el valor de la señal de asentimiento enviada durante el noveno ciclo de SCL. Esto permite identificar si el dispositivo maestro o esclavo (dependiendo de si la operación es de lectura o escritura respectivamente) ha recibido e interpretado el byte adecuadamente. En caso de que se esté recibiendo el primer byte de un mensaje (dirección del esclavo y tipo de operación), esta señal indicará si el dispositivo esclavo ha sido correctamente direccionado. A continuación, el sistema espera la recepción de un nuevo byte o una condición de stop o reinicio. Si se detecta un nuevo byte, el sistema pondrá a nivel bajo su salida de EOM para indicar al bloque de control que el bit capturado no era el último de un mensaje y pasará el valor de la señal de asentimiento a su salida ACK. Por último activará su salida de NEW_EOM para indicar al bloque de control que almacene dicha información. Si tras recibir una señal de asentimiento se identifica una condición de parada o reinicio, el sistema actuará de forma similar a la expuesta anteriormente, pero en este caso la salida EOM se pondrá a nivel alto para indicar al bloque de control que el último byte recibido era el final de un mensaje.

Además, el sistema incluye una salida que se conectará a un LED para indicar que se está capturando un mensaje de modo que si la transmisión se corta el LED permanecerá encendido indicando que la transmisión se ha interrumpido sin finalizarse. Por el contrario, si la transmisión finaliza antes de lo debido (se recibe una condición de reinicio o parada mientras se está capturando un byte) el sistema enviará al ordenador el byte capturado presentando ceros en las posiciones que no han sido recibidas e indicando que la señal de asentimiento no se ha recibido por lo que se podrá saber que la transmisión se ha interrumpido.

Implementación:

El diseño de este bloque incluye diversos elementos que permiten realizar la captura de forma adecuada:

- Un sincronizador se conecta a una de las entradas provenientes del bus I2C para que éstas sean adaptadas al dominio del reloj del sistema y poder sincronizar todo el diseño. Esto se consigue mediante un doble registro (dos biestables) conectado a cada entrada que en cada ciclo del reloj copian el valor de la línea de datos (SDA) y del reloj de sincronización (SCL) a su salida. El segundo registro copia el valor de la salida del primero con lo que se evitan problemas en el sistema como la metaestabilidad.
- Unos detectores de flanco se encargan de identificar las subidas y bajadas de la línea de datos para poder identificar las condiciones de inicio, reinicio y fin de un mensaje. Otro detector identifica los flancos de subida de SCL para que se capture cada bit en el momento adecuado.
- Un contador de 4 bits permite identificar el número de ciclos de SCL que han transcurrido durante el envío del byte. Este contador aumenta su valor cada vez que se captura un bit o una señal de asentimiento y es controlado desde la máquina de estados implementada. Además, este contador incluye una entrada de reset que permite poner el valor a 0 para recibir un nuevo byte.
- Se emplea un registro de desplazamiento de 8 bits para almacenar el byte capturado de la línea SDA. Este registro posee una señal de habilitación de recepción que permite desplazar el registro una posición hacia la izquierda y que el bit capturado se sitúe en la posición más baja del registro (LSB). De este modo se consigue que el byte capturado, que se envía de MSB a LSB como se detalló en la sección 2.2.3, se almacene en el registro de la misma manera que se envió, facilitando la gestión del envío de los datos al ordenador.
- Por último, el diseño está controlado mediante una FSM cuyos estados y transiciones entre ellos se muestran y definen a continuación:

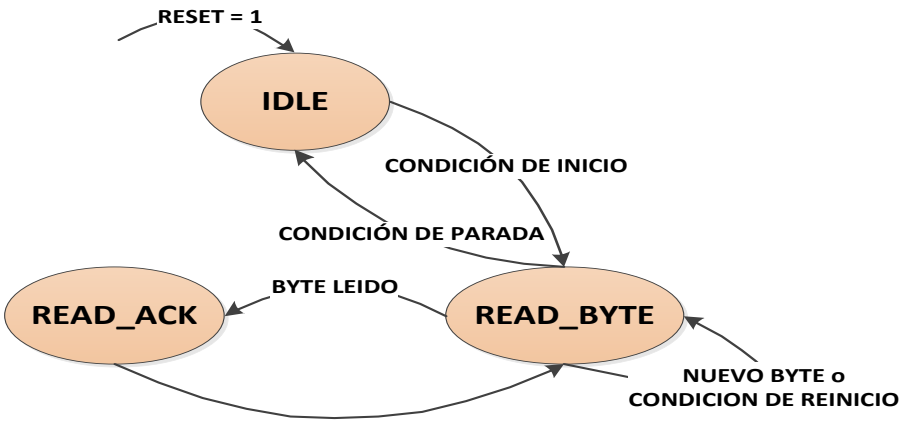


Figura 37: FSM del bloque I2C_Sampler

Los estados mostrados tienen la siguiente finalidad:

- Idle: Estado de reposo del sistema, el LED de RUNNING permanece apagado indicando que no están capturando datos. Cuando se detecta una condición de inicio de mensaje el sistema pasa al estado read_byte para capturar los datos.

- Read_byte: En este estado se capturan cada uno de los bits del byte transferido y se identifica si éste era el final de un mensaje o no. Cada vez que se identifica un flanco de subida en el reloj SCL se almacena el valor de la línea SDA en el registro de 8 bits mediante la activación de su señal de desplazamiento de bits y se aumenta el valor del contador de bits para calcular el número de estos capturados. Por otro lado, se identifica si el byte recibido corresponde al final de un mensaje o si se ha producido un corte en la transmisión mediante varias condiciones:
    - Se identifica una condición de inicio: si el contador de bits recibidos está a 0, el byte que se acaba de recibir era el final de un mensaje por lo que la salida de EOM se fija a 1 para indicarlo y el sistema reinicia el contador de bits para comenzar el análisis de un nuevo mensaje permaneciendo en este estado ya que directamente comienza la transmisión del nuevo mensaje. La salida NEW_EOM se pone a nivel alto para que el bloque de control guarde el valor de la señal de asentimiento y la información de final de mensaje. si el contador de bits no está a 0 cuando se identifica el reinicio, se indicará que la transmisión se ha cortado durante el envío ya que la salida de datos contendrá los bits recibidos y ceros en las posiciones no recibidas. Además, la señal de asentimiento se fijará a nivel alto indicando que el receptor no ha recibido el byte completo. La salida EOM se pondrá a nivel alto para indicar que el byte recibido es el final del mensaje y se almacenará la información poniendo a nivel alto la salida NEW_EOM.
    - Se identifica una condición de stop: de nuevo se evalúa el valor del contador de bits recibidos para saber si el stop ha llegado tras un byte completo o durante la recepción de éste. Si el contador está a 0, el último byte recibido era el final de un mensaje por lo que también se activa la salida de NEW_EOM y se fija la salida EOM a nivel alto. Si el contador tiene un valor distinto de 0, el stop se ha recibido mientras se estaba enviando un byte por lo que, como en el caso anterior, la salida de datos contendrá los bits capturados y ceros en las posiciones no recibidas. La salida ACK estará fijada a nivel alto al igual que la salida EOM, para indicar que el byte enviado es el final de un mensaje y que no se ha recibido señal de asentimiento por lo que la transmisión ha sido interrumpida. En cualquiera de estos casos, el sistema reinicia el contador pero pasa al estado de reposo y esperar a que se identifique una condición de inicio que implique que se va a transmitir un nuevo mensaje.
    - Si el contador de bits vale 1 y no se está recibiendo el primer byte del mensaje: Indica que se ha recibido ya el primer bit de otro byte por lo que el byte recibido anteriormente no era el final del mensaje. La salida EOM se fija a cero para indicarlo y se activa la salida NEW_EOM para que la información se almacene.

  Cuando un byte se ha recibido completamente el contador lo identifica poniendo a nivel alto una señal que es evaluada por la FSM. Si se detecta que se han recibido los 8 bits el sistema pasa al estado read_ack para capturar la señal de asentimiento del byte almacenado.

- Read_ack: En este estado se almacena el valor de la señal de asentimiento enviada por el maestro o el esclavo cuando se detecta el noveno ciclo de SCLK. Tras esta detección, el sistema vuelve al estado de read_byte para capturar un nuevo byte e identificar si el anterior era el final del mensaje o no como se ha explicado previamente.

3.3.3.3. Bloque de control I2C

De manera análoga al analizador de protocolos SPI, se ha implementado un bloque capaz de almacenar los datos capturados y gestionar su envío mediante el puerto serie. Para facilitar el desarrollo de este bloque y considerando que algunas de las gestiones a realizar con los dato son semejantes a las realizadas en el protocolo SPI, se han reutilizado algunos de los bloques diseñados y descritos previamente.

En esta sección se detallará el Interfaz que emplea el diseño para comunicar con el exterior, su funcionalidad y la implementación que se ha realizado para alcanzarla. Posteriormente se describirán los bloques contenidos en él, haciéndose únicamente una pequeña descripción de su funcionalidad y los elementos a los que conecta ya que, dado que son los mismos bloques empleados en el protocolo SPI, el resto de información puede ser consultada en los apartados anteriores.

En primer lugar, la siguiente imagen refleja la lista de entradas y salidas que posee el bloque Control_I2C que serán descritas posteriormente:

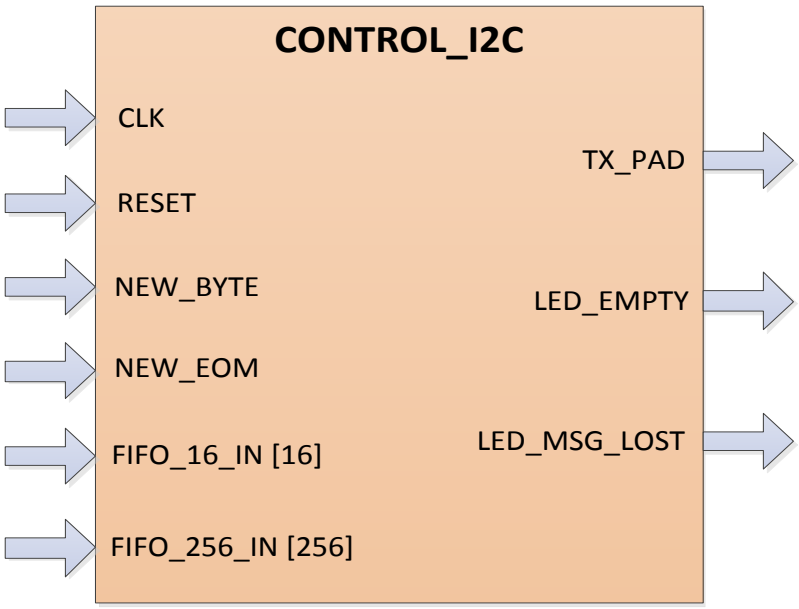


Figura 38: Interfaz del bloque de control I2C

Interfaz del diseño (entradas/salidas)

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
CLK	Entrada	CLK	1	Flanco de subida		Entrada del reloj general del sistema

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
RESET	Entrada	RESET	1	‘1’		Puerto para reiniciar el componente
NEW_BYTE	Entrada	n_byte	1	‘1’		Entrada proveniente del bloque I2C_Sampler que indica que hay un nuevo byte que almacenar
NEW_EOM	Entrada	n_eom	1	‘1’		Entrada proveniente del bloque I2C_Sampler que indica que hay información de final de mensaje que escribir
FIFO_8_IN	Entrada	byte_rec	8	No aplica		Entrada de datos de la memoria FIFO de 8 bits
FIFO_2_IN	Entrada	ack_eom	2	No aplica		Entrada de datos de la memoria de 2 bits
TX_PAD	Salida	TX_PAD	1		‘1’	Salida del transmisor serie para efectuar los envíos de datos al ordenador

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
LED_EMPTY	Salida	LED_EMPTY	1		'1'	Puerto que indica que las memorias FIFO están vacías
LED_MSG_LOST	Salida	LED_MSG_LOST	1		'0'	Terminal que se conectará a un LED para identificar la pérdida de mensajes capturados.

Tabla 22: Interfaz del bloque Control I2C

Funcionalidad

El módulo descrito en este apartado se encarga de almacenar la información capturada y manipularla para que se envíe al ordenador con una estructura sencilla que permita su análisis posterior. El bloque almacena cada uno de los bytes capturados en una memoria FIFO de 8 bits de manera que, cuando el mensaje se va a enviar al ordenador, el primer byte que se transfiera corresponderá al primer byte recibido de un mensaje y el resto se mostrarán en el orden en el que han sido recibidos. Además, otra memoria de 2 bits es la encargada de almacenar el valor de la señal de asentimiento y un bit que indica si el byte al que corresponde dicha señal de asentimiento era el final de un mensaje o no. Esto nos permite recomponer la estructura de un mensaje completo haciendo que en la pantalla se muestre cada mensaje con todos los bytes que contiene en una línea.

Cuando el dispositivo de captura indica que ha recibido un byte, el sistema se encarga de almacenarlo en la memoria correspondiente si esta no está llena. Si la memoria está llena el sistema activará su salida LED_MSG_LOST para indicar que un byte ha sido capturado pero no ha podido ser almacenado por falta de capacidad. Este LED permanecerá encendido aunque mensajes posteriores sí que puedan ser almacenados para indicar que el conjunto de mensajes recibidos en el ordenador no contiene todos los mensajes que han sido capturados y por tanto transmitidos a través del bus I2C. Como caso especial, cabe indicar que la dirección del esclavo direccionado (compuesta de 7 bits como se ha indicado previamente en la sección 2.2.3) será almacenada junto al tipo de operación (lectura o escritura) como un byte.

Cuando el sistema de captura indica que ha recibido la señal de asentimiento de un byte y que ha identificado si dicho byte era el final de un mensaje, la entrada NEW_EOM estará a nivel alto. En ese momento, el dispositivo de control almacenará dicha información en una memoria FIFO de 2 bits de modo que la información correspondiente al primer byte será la primera en

leerse y así sucesivamente con el resto de bytes permitiendo recomponer el mensaje de una manera adecuada.

Para enviar los mensajes al ordenador, el sistema identifica si las memorias contienen datos. En caso positivo, el diseño controla el transmisor serie para que envíe una cabecera que permita identificar al mensaje, asignándole un número de mensaje transmitido. Posteriormente, el sistema envía una cabecera que indica que se va a transmitir la dirección del esclavo direccionado y el tipo de operación que se va a realizar, pudiendo ser esta de lectura o escritura. Completado este paso, se lee la memoria de bytes para extraer la información capturada.

Cuando se lee un byte de la memoria de datos, los datos obtenidos son almacenados en un registro de 8 bits, cuyos 4 primeros bits se conectan al conversor de hexadecimal a ASCII. De esta manera se consigue obtener el equivalente de 8 bits correspondiente al carácter ASCII que representa el carácter hexadecimal capturado. Cuando dicho valor ha sido enviado por el puerto serie, el diseño desplaza 4 bits el registro de 8 bits para enviar la parte baja del byte de manera similar a como se ha transferido la parte alta.

En el caso de la dirección del esclavo, se envían 2 caracteres hexadecimales (8 bits) correspondientes a los 7 bits de dirección del esclavo y el tipo de operación a realizar.

Finalizada la transmisión de la dirección, el sistema envía una nueva cabecera que remarca el tipo de operación solicitada, enviando una R (Read) cuando la operación es de lectura y una W (Write) cuando la operación es de escritura. A continuación, el sistema enviará una cabecera seguida del valor de la señal de asentimiento del esclavo que permitirá saber si el direccionamiento se ha realizado de forma adecuada. Si el esclavo ha sido direccionado, se enviará un 0 ya que éste lo identifica mediante el envío de un nivel bajo durante el noveno ciclo de la transmisión. En caso de que el esclavo no haya sido direccionado, la línea SDA permanecerá a nivel alto durante ese noveno ciclo de SCL por lo que se reflejará un 1 en la pantalla indicando que no se ha direccionado a ningún esclavo.

Tras esto, comienza la transmisión de los bytes del mensaje capturado. Para ello, se lee un nuevo byte de la memoria y se realiza la conversión explicada anteriormente para que se envíe como 2 caracteres hexadecimales al ordenador.

A continuación, el sistema envía de nuevo una cabecera que identifica la señal de asentimiento del byte mostrado y lee la memoria de 2 bits para obtener el valor de ésta. Si el receptor ha enviado la señal de asentimiento el sistema enviará al ordenador un 0 ya que esta señal se identifica con un nivel bajo de tensión. Si el receptor no ha enviado señal de asentimiento, el ordenador reflejará un 1 indicando que había un nivel alto en la línea SDA y que por tanto el receptor no ha enviado señal de asentimiento.

Tras esto, se comprueba el valor del segundo bit almacenado en la memoria de 2 bits, que corresponde a la información de final de mensaje del byte transferido. Si este bit es cero, se sabrá que el byte transmitido no era el final de un mensaje por lo que se enviará al ordenador un carácter de separación de bytes y se repetirá el proceso para enviar otro byte contenido en el mismo mensaje. Por el contrario, si el bit de final de mensaje es 1 el sistema reconocerá que el byte que ha transmitido era el último contenido en un mensaje por lo que enviará a través

del puerto serie los caracteres de final de mensaje y retorno de carro para que siguiente mensaje a transmitir se muestre en la pantalla del ordenador en una nueva línea.

Así se consigue que cada mensaje, independientemente del número de bytes que contenga, se muestre en una línea en la pantalla facilitando el análisis posterior de los datos recibidos.

Considerando lo anteriormente expuesto, los mensajes reflejados en el ordenador presentarán la estructura reflejada en la siguiente imagen:

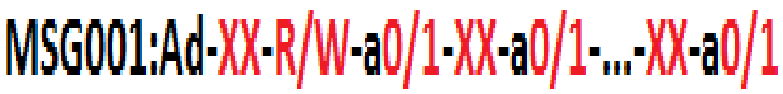


Figura 39: Representación general de un mensaje I2C

Donde X representa cada uno de los posibles caracteres hexadecimal que corresponden a los 4 bits más significativos del byte capturado o a los 4 bits menos significativos. R/W representa el tipo de operación, de lectura (R) o de escritura (W). 0/1 corresponde al valor nivel lógico de la señal de asentimiento, siendo éste 0 cuando el receptor asiente la transmisión o 1en caso de que no lo haga.

A modo de ejemplo, si el maestro I2C direcciona a un esclavo cuya dirección es “1010000” y solicita una operación de escritura (bit 8 = 1) enviando 2 bytes, que serán “00001011” y “11110110” y el esclavo asiente su recepción, el mensaje que se observará en la pantalla del ordenador será:

Figura 40: Representación del ejemplo de mensaje I2C

Implementación:

Dada la complejidad de este sistema, se ha optado por dividir la implementación en diversos bloques encargados de realizar una función específica. Todos estos bloques se conectan entre sí para conseguir que cada uno de ellos funcione adecuadamente mediante su control, que es realizado por la máquina de estados que se describirá posteriormente. Este diseño incluye, además de componentes, una serie de procesos que se encargar de gestionar el control de los bloques empleados. Los bloques empleados en este diseño junto a los procesos y demás elementos empleado para su control pueden observarse en la imagen expuesta a continuación y serán descritos posteriormente. Además, pueden observarse las principales conexiones empleadas entre dichos bloques:

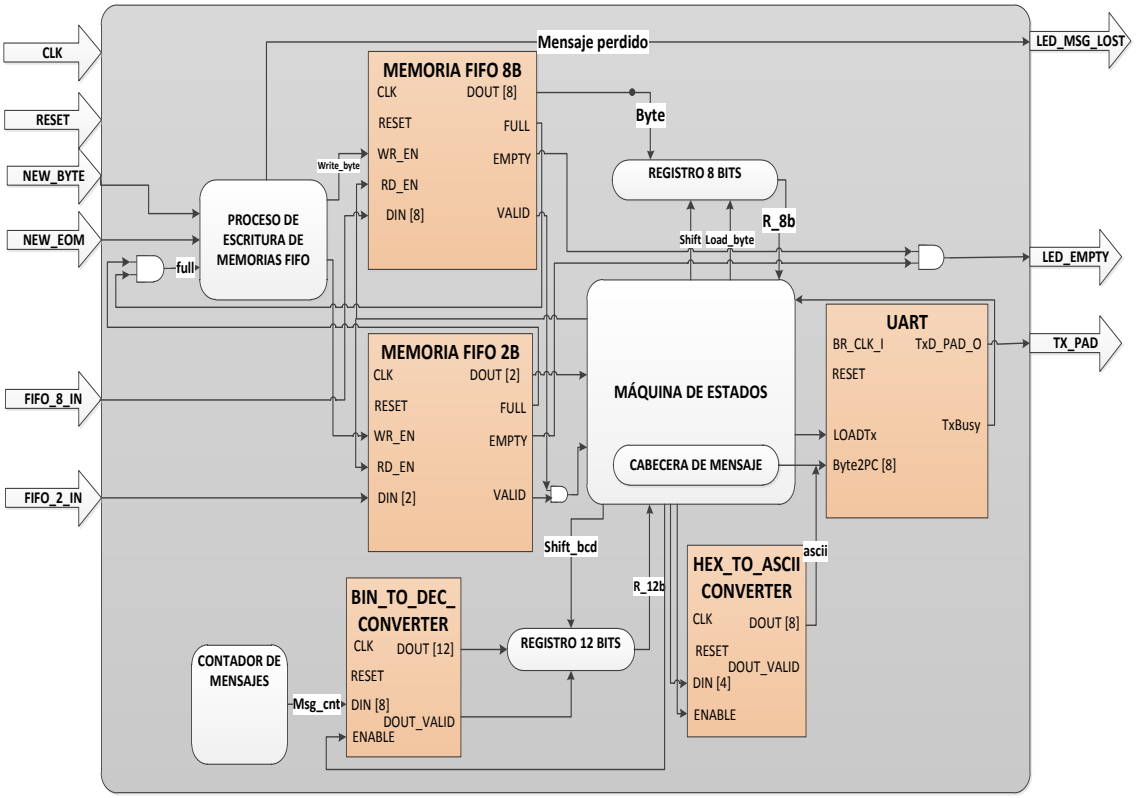


Figura 41: Diagrama de bloques del componente Control I2C

Como puede observarse en el diagrama expuesto arriba, el diseño incluye diversos componentes:

- Un proceso se encarga de realizar la escritura de las memorias en el momento adecuado. Cuando se detecta un nivel alto en la entrada NEW_BYTE, el contenido de la entrada FIFO_8_IN es almacenado en la memoria de 8 bits para que el byte capturado quede guardado. Si se detecta un nivel alto en la entrada NEW_EOM, el valor de la señal de asentimiento de recepción de un byte y el bit que refleja si dicho byte era el final de un mensaje son almacenados en la memoria de 2 bits.
- Un contador de 8 bits, controlado desde una máquina de estados, permite calcular el número de mensajes que han sido transmitidos al ordenador. Este contador incluye una entrada que permite aumentar su valor cada vez que el sistema ha terminado de enviar un mensaje.
- Otro contador descendente, en este caso de 2 bits, permite conocer el número de caracteres BCD que se han enviado (centenas, decenas y unidades) correspondientes al número de mensaje que se está transfiriendo al ordenador. Este contador incluye una entrada que permite disminuir su valor y otra empleada para fijar el valor inicial de cuenta a 2 haciendo posible saber a la máquina de estados que carácter BCD debe enviar.
- Por último, un contador ascendente de 4 bits permite calcular el carácter de cabecera que debe ser transferido en cada momento.
- Un registro de 8 bits es empleado para almacenar los bytes leídos de la memoria de 8 bits. Este registro incluye una señal que habilita la carga de los datos y otra que permite realizar un desplazamiento de 4 posiciones a la izquierda en el registro. Esta última señal es controlada desde la FSM y permite que se envíen al conversor de hexadecimal a ASCII los 4

bits más altos o más bajos del byte a transmitir para que se codifiquen de forma adecuada y puedan ser transferidos al ordenador dónde se representarán como un carácter hexadecimal.

- Otro registro de12 bits se encarga de almacenar el resultado de la conversión a BCD del número de mensaje enviado. Este registro carga la salida del conversor mediante una señal que controla la máquina de estados y posee otra entrada que permite realizar un desplazamiento de 4 posiciones a la izquierda para que se envíen al conversor de hexadecimal a ASCII los 4 bits correspondientes a cada uno de los caracteres BCD (centenas, decenas o unidades). Esta última señal es necesaria ya que la entrada del conversor está conectada únicamente a los 4 primeros bits del registro por lo que hay que desplazar los bits para enviar cada uno de los caracteres.
- Por último, una máquina de estados controla todos los demás componentes creados para el diseño. Esta FSM incluye diversos estados y transiciones entre ellos que pueden ser observados en la siguiente imagen y que serán detallados a continuación de ésta:

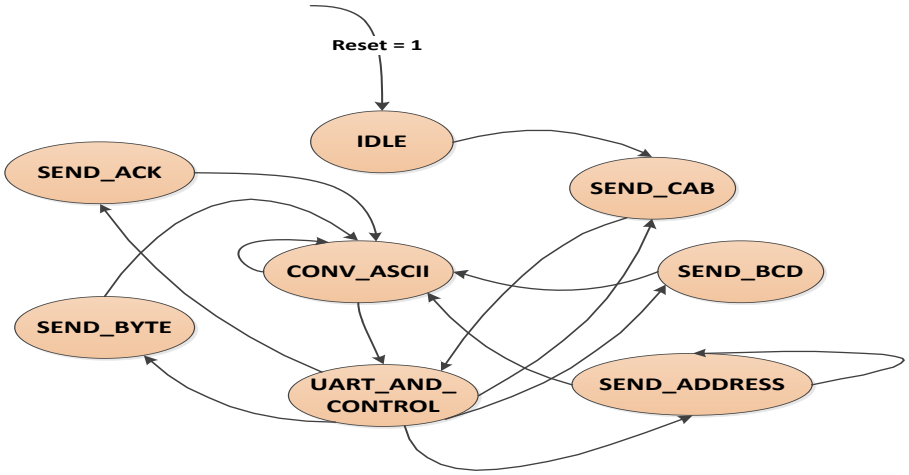


Figura 42: FSM del bloque de control I2C

- Idle: Este es el estado de reposo del sistema y se entrará en él cuando se detecte que las memorias no tienen información almacenada para enviar. Si las memorias tienen datos disponibles, se pasará al estado send_cab para comenzar la transmisión del mensaje.
- Send_cab: Este estado se encarga de activar el transmisor serie incluido en la UART y facilitarle el carácter ASCII correspondiente a la posición de la cabecera que se desea transferir. Mediante la evaluación del contador de posiciones de cabecera se selecciona el carácter que se debe transmitir en cada momento. Esta implementación permite que los valores correspondientes a los caracteres ASCII de la cabecera queden almacenados en una memoria ROM de forma que el valor del contador de posiciones se emplee como dirección a leer de la memoria reduciendo el uso de recursos lógicos de la implementación. Desde este estado se pasa al estado uart_and_control donde, evaluando la señal de control r_control como se detallará posteriormente, se decide el siguiente estado para continuar la transmisión.
- Send_bcd: Este estado se emplea para controlar la conversión y el envío del número de mensaje transmitido al ordenador. Los 4 bits más significativos del registro de 12 bits que contiene el valor BCD del número de mensaje son enviados al conversor de hexadecimal a

ASCII para que posteriormente sean enviados al ordenador de forma adecuada. Evaluando el valor del contador auxiliar se comprueba si se han enviado las centenas o las decenas, estableciendo el retorno posterior a este estado mediante la señal r_control para enviar el siguiente carácter BCD. Si el contador tiene el valor 0, el carácter correspondiente a las unidades habrá sido enviado por lo que la señal r_control identificará que debe pasarse al estado send_cab para enviar la cabecera de dirección del mensaje.

- Send_address: este estado es el encargado de transferir al conversor de hexadecimal a ASCII la parte alta o baja de dirección del esclavo. Mediante la evaluación de una señal que indica si se ha trasferido la parte alta de un byte (4 bits más significativos) o la parte baja (4 bits menos significativos) se definen diferentes acciones:
    - Si la parte alta no ha sido enviada: Se envían al conversor los 4 bits correspondientes y se activa la señal de desplazamiento del registro de 8 bits para que los 4 bits siguientes estén disponibles para su envío posterior. Se fija el valor adecuado en la señal r_control para conseguir que se retorne a este estado para enviar la parte baja del byte.
    - Si la parte alta ya ha sido enviada: se envían al conversor los 4 bits de la parte baja del byte y se establece el valor de la señal r_control para que se pase a enviar la cabecera que identifica tipo de operación realizado (lectura o escritura).
- Conv_ASCII: En este estado se espera a que el conversor de hexadecimal a ASCII haya finalizado la conversión para enviar el resultado al transmisor UART. Cuando la conversión finaliza se pasa al estado UART_and_control para definir el siguiente estado.
- Send_byte: Este estado es semejante al estado send_address pero su función es controlar la transmisión de un byte de datos capturado en el mensaje. De nuevo, una señal de control indica si se va a transferir la parte alta o baja de un byte y mediante su evaluación se definen las acciones a realizar:
    - Si no se ha enviado la parte alta: Ésta será transferida al conversor de hexadecimal a ASCII para que posteriormente sea enviada al ordenador como un carácter hexadecimal y se fijará la señal r_control para volver después a este estado y enviar la parte baja del byte. El registro de8 bits será desplazado a la izquierda para tener disponibles en su parte alta los datos que se deben enviar posteriormente.
    - Si la parte alta ya ha enviada: Se mandará al conversor la parte baja del byte a transmitir y se fijará señal de control para que posteriormente se envíe el valor de la señal de asentimiento del byte manipulado.
- Send_Ack: Este estado ha sido diseñado para enviar al conversor de hexadecimal a ASCII el valor de la señal de asentimiento del byte que se acaba de transferir. Una vez que el valor ha sido leído de la memoria de 2 bits, se envía al conversor un grupo de 4 bits formado por 3 ceros y el valor de la señal de asentimiento situado en la posición más baja. De esta manera se consigue que el conversor reciba “0000” o “0001” y se envíe al ordenador el 0 o 1 correspondiente al valor de la señal de asentimiento como se ha expresado en la definición del mensaje I2C. Además, este estado se encarga de evaluar si la señal de asentimiento corresponde al último byte de un mensaje o no. Para ello, el bit menos significativo de la salida de la memoria de 2 bits es evaluado y, en función de su valor, se

asigna un valor distinto a la señal r_control para que se finalice la transmisión del mensaje si el byte enviado era el final de éste o se pase a enviar la cabecera de un nuevo byte si no lo era.

- UART_and_control: Este estado es el encargado de esperar a que el transmisor UART esté libre para recibir un nuevo sato a enviar y decidir el siguiente estado para continuar la transferencia del mensaje. mediante un case, se evalúa el valor de la señal r_control para definir el siguiente estado de la FSM como refleja la tabla expuesta a continuación:

Valor de r_control	**Interpretación del valor y acciones realizadas**
0000	Se ha enviado un carácter de cabecera y debe enviarse otro más. Se incrementa el contador de posiciones de cabecera.
0001	Se ha enviado un carácter de cabecera y debe enviarse el primer digito BCD del número de mensaje. Se activa el conversor a BCD y, cuando la conversión ha finalizado, se pasa al estado send_bcd cargándose en el contador auxiliar el valor 2.
0010	Se ha enviado un carácter BCD y debe enviarse otro. Se desplaza el registro de 12 bits para poder convertir el siguiente dígito, se resta 1 al contador auxiliar y se pasa al estado send_bcd.
0011	Se ha enviado el dígito BCD de las unidades y debe enviarse la cabecera de la dirección del esclavo. Se aumenta el contador de posiciones de cabecera y se pasa al estado send_cab.
0100	Se ha enviado la cabecera de la dirección del esclavo y debe enviarse la parte alta de este byte. Para ello, se lee la memoria FIFO de 8 bits y se almacena su salida en el registro de 8 bits implementado. Se pasa al estado send_address para comenzar el envío de la dirección.
0101	Se ha enviado la parte alta de la cabecera del mensaje por lo que se retorna al estado send_address para enviar la parte baja.
0110	La dirección del esclavo ha sido enviada completamente por lo que se debe pasar al estado send_cab para transferir la cabecera de la señal de asentimiento del direccionamiento.
0111	La cabecera de la señal de asentimiento de la dirección ha sido transferida por lo que debe enviarse el valor de ésta. Dicho valor está almacenado en la salida de la memoria de 2 bits empleada que, para obtener la correspondencia entre el byte enviado, su *Acknowledge* y la información de si éste es final de mensaje, ha sido leída a la vez que la memoria que contiene el byte a transmitir.
1000	Se ha enviado el valor de la señal de asentimiento de un byte y se ha identificado que el mensaje no ha terminado. Se pasa al estado send_cab para comenzar la transmisión de un nuevo byte.

Valor de r_control	Interpretación del valor y acciones realizadas
1001	Se ha enviado el valor de la señal de asentimiento de un byte y se ha identificado que se trataba del último byte contenido en el mensaje. Se envía el carácter ASCII de final de línea y el de retorno de carro por el puerto serie para que el ordenador se prepare para recibir más mensajes.
1010	Se ha enviado el carácter de separación de bytes y debe enviarse un nuevo byte contenido en el mensaje. Se pasa al estado send_byte para que éste sea transferido.
1011	Se ha enviado un byte de datos completo y debe enviarse la cabecera que indicará su señal de asentimiento. Se pasa al estado send_cab para llevar a cabo esta acción.
1100	Debe enviarse un nuevo byte de mensaje por lo que las memorias, tanto la que contiene el byte a enviar como la que contiene su señal de asentimiento y la información de final de mensaje, son leídas. Cuando las salidas de datos de las memorias son válidas, se pasa al estado send_byte para comenzar la transmisión de los datos.
1101	Valor no empleado. Este valor no se asigna nunca a la señal pero, en caso de que se alcanzase, el sistema volvería al estado de reposo para evitar problemas no contemplados.
1110	Valor no empleado. Este valor no se asigna nunca a la señal pero, en caso de que se alcanzase, el sistema volvería al estado de reposo para evitar problemas no contemplados.
1111	El mensaje se ha enviado completo por lo que se pasa al estado de reposo para comprobar si hay más mensajes que transmitir. El contador de mensajes incrementa su valor para controlar el número de mensajes transmitidos.

Tabla 23: Interpretación de la señal r_control en el bloque de control I2C

3.3.3.4. FIFO

Como se ha comentado anteriormente, este diseño está contenido dentro del bloque de control I2C y se encarga de almacenar la información capturada a través del bus I2C.

Dado que el diseño empleado es el mismo que el usado para el analizador SPI (ver sección 3.3.2.4) este diseño no se describirá en profundidad. Únicamente se expondrá la lista de señales a la que se conecta su interfaz y se describirá la funcionalidad para la que han sido diseñados. Por último, se indicarán las características de implementación que difieren respecto al diseño empleado en el analizador SPI.

Descripción de señales a las que conecta la interfaz del bloque:

La siguiente tabla refleja loa lista de puertos disponibles en el diseño y las señales a las que se conectan éstos. Dado que el diseño emplea dos memorias con interfaz semejante pero

diferente tamaño (memoria FIFO 8b y memoria FIFO 2b), la tabla incluye las señales a las que se conectan ambos bloques:

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
CLK	Entrada	CLK	1	Flanco de subida		Entrada del reloj general del sistema
RST	Entrada	RESET	1	‘1’		Puerto para reiniciar el componente
DIN	Entrada	FIFO_8_IN o FIFO_2_IN	8 o 2	No aplica		Entrada de datos de la memoria
WR_EN	Entrada	Wr_en_8 O Wr_en_2	1	‘1’		Puerto de habilitación de escritura
RD_EN	Entrada	Rd_en	1	‘1’		Puerto de habilitación de lectura
DOUT	Salida	FIFO_dout_8 o FIFO_dout_2	16 o 256		Todos los bits a 0	Salida de datos de la memoria
FULL	Salida	full_8 o full_2	1		‘0’	Salida que indica que la memoria está llena
WR_ACK	Salida	wr_ack_8 o wr_ack_2	1		‘0’	Puerto que indica que la escritura ha sido correcta
EMPTY	Salida	empty_8 o empty_2			‘1’	Salida que indica que la memoria está vacía, no contiene datos

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
VALID	Salida	valid_8 o valid_2			'0'	Puerto que indica que la salida de datos es válida y ésta puede ser leída.

Tabla 24: Interfaz de las memorias FIFO del analizador I2C

Funcionalidad:

Estas memorias son empleadas para almacenar los datos capturados del mensaje de forma que puedan ser accesibles en el momento que van a ser transferidos al ordenador. Este diseño permite que la transmisión de datos por el protocolo I2C sea mucho más rápida que el envío de estos al ordenador y que no se pierda la información capturada. Dado que se han implementado 2 diseños similares, cada uno posee su propia funcionalidad:

- Memoria de 8 bits: se encarga de almacenar cada uno de los byte capturados, ya sean la dirección de un esclavo o un byte de datos.
- Memoria de 2 bits: permite almacenar el valor de la señal de asentimiento correspondiente a cada uno de los bytes de la memoria anterior y un bit que indica si éste era el último byte contenido en un mensaje o no.

La escritura de ambos dispositivos se realizada en momentos distintos ya que la información a almacenar no está disponible en el mismo momento. Sin embargo, la lectura se realizará de forma simultánea para que, gracias al funcionamiento de una memoria FIFO ya explicado, se disponga en el mismo momento del byte a transmitir y la información relacionada con él.

Implementación:

Para la implementación de estos dispositivos se ha usado el asistente disponible en el entorno de desarrollo usado de manera análoga a la empleada en el analizador SPI. Únicamente se ha variado el parámetro que define el ancho de entrada de las memorias, los cuales han sido configurados para aceptar 8 o 2 bits en función del diseño implementado. Para observar la configuración completa empleada puede revisarse el apartado de implementación de la sección 3.3.2.4.

3.3.3.5. UART

Este bloque, contenido en el diseño de control I2C, es semejante al empleado en el analizador SPI. Las características principales de este diseño no se describirán ya que aparecen reflejas previamente en la sección 3.3.2.7. a continuación se presentarán una serie de tablas donde puede observarse la interfaz empleada por el diseño y las señales a las que se conectarán en este caso los puertos:

Interfaz del diseño (entradas/salidas):

Las tablas mostradas a continuación recogen los datos característicos del genérico y cada uno de los puertos del bloque UART:

- Genéricos:

Genérico	Tipo	Valor asignado	Descripción
BRDIVISOR	Entero	1302 (9600 baudios)	Parámetro empleado para definir la velocidad de transmisión que empleará el puerto serie. Pueden seleccionarse valores desde 0 hasta 65535 para obtener la velocidad deseada como se explicará posteriormente

Tabla 25: Genéricos de la entidad UART

- Puertos:

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
BR_Clk_I	Entrada	CLK	1	Flanco de subida		Entrada del reloj de sistema
RESET	Entrada	RESET	1	‘1’		Puerto para reiniciar el componente
Rx_PAD_I	Entrada	'1'	1	‘0’		Entrada de recepción serie RS232
LoadTx	Entrada	load_tx	1	‘1’		Puerto para habilitar una transmisión
Byte2PC	entrada	No conectado	8	No aplica		Vector que contiene el valor ASCII (en binario) que se desea enviar.
TxD_PAD_O	Salida	TX_PAD	1		‘1’	Salida de transmisión serie RS232

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
ByteFromPC	Salida	r_data2tx	8		Todos los bits a 0	Vector que contiene el valor ASCII recibido.
TxBusy	Salida	tx_busy	1		'0'	Salida que indica que el módulo está ocupado
RxAv	Salida	No conectado	1		'0'	Salida que indica que se ha recibido un nuevo byte

Tabla 26: Interfaz del módulo UART del analizador I2C

Funcionalidad:

En este caso, la funcionalidad de este diseño es transferir mediante el puerto serie los mensajes capturados de forma que puedan ser interpretados en el ordenador de destino. Este bloque se encarga de transferir la información según establece el protocolo RS-232 enviando cadenas de 8 bits que serán interpretadas en el ordenador como un carácter ASCII.

3.3.3.6. Conversor de binario a BCD

El diseño del conversor de binario a BCD, que está contenido en el bloque de control I2C, ya ha sido presentado previamente ya que se trata del mismo diseño empleado en el analizador SPI. Sus características de diseño pueden ser consultadas en la sección 3.3.2.6. Las únicas diferencias con diseño ya presentado son las señales a las que se conectan cada uno de los puertos presentes y que son definidas en la tabla expuesta a continuación:

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
CLK	Entrada	CLK	1	Flanco de subida		Entrada del reloj de sistema
RST	Entrada	RESET	1	'1'		Puerto para reiniciar el componente
DIN	Entrada	r_data2tx	8	No aplica		Entrada de datos a convertir

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
ENABLE	Entrada	enable_bcd	1	'1'		Puerto para habilitar la conversión
DOUT	Salida	r_bcd_o	12		Todos los bits a 0	Salida del valor convertido. Cada grupo de 4 bits representa un carácter BCD
DOUT_VALID	Salida	bcd_valid	1		'0'	Salida que indica que la conversión ha finalizado

Tabla 27: Interfaz del conversor de binario a BCD del analizador I2C

Funcionalidad:

En este caso, el diseño ha sido empleado únicamente para obtener el valor, codificado en BCD, del número de mensaje I2C a transmitir. Esto permite que en el ordenador se muestre adecuadamente el valor como se ha descrito previamente.

3.3.3.7. Conversor de Hexadecimal a ASCII

Este módulo se encarga de convertir un grupo de 4 bits, el cual es considerado como un carácter hexadecimal, en el equivalente ASCII a dicho carácter hexadecimal.

Dado que este bloque ya fue utilizado en el analizador SPI, no se detallaran sus características ya que éstas se pueden consultar en la sección 3.3.2.5. Únicamente se describirá la lista de señales a las que se conectan cada uno de los puertos que posee y se describirá su funcionalidad específica dentro del bloque de control I2C.

La interfaz de este bloque queda reflejada en la siguiente tabla:

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
CLK	Entrada	CLK	1	Flanco subida		Entrada del reloj común
RST	Entrada	RESET	1	'1'		Puerto para reiniciar el componente

Puerto	Tipo	Señal a la que conecta	Número de bits	Nivel activo	Valor de Reset	Descripción
DIN	Entrada	r_data2conv	4	No aplica		Entrada de datos a convertir
ENABLE	Entrada	enable_conv	1	'1'		Puerto para habilitar la conversión
DOUT	Salida	r_data_conv_o	8		Todos los bits a 0	Salida del valor convertido. Valor binario del carácter ASCII
DOUT_VALID	Salida	conv_valid	1		'0'	Salida que indica que la conversión ha finalizado

Tabla 28: Interfaz del conversor de hexadecimal a ASCII del analizador I2C

Funcionalidad:

En este caso específico, el conversor de hexadecimal a ASCII se emplea para obtener el vector de 8 bits que contiene el equivalente ASCII del grupo de 4 bits del mensaje que se desea transmitir al ordenador. Dicho grupo de 4 bits puede corresponder a la parte alta (4 bits más significativos) o a la parte baja (4 bits menos significativos) de un byte capturado a través de la línea SDA del protocolo I2C. De esta manera se consigue que el byte capturado aparezca reflejado en el ordenador como un conjunto de 2 caracteres hexadecimales (8 bits) facilitándose el análisis de los datos capturados.

4. Verificación

Una vez desarrollado el diseño es necesario verificar su correcto funcionamiento mediante diversas técnicas que nos permitan realizar un análisis a diferentes niveles.

En primer lugar, será necesario realizar diferentes pruebas de simulación a cada uno de los bloques implementados para corroborar el adecuado comportamiento de cada una de las señales empleadas en el diseño.

Posteriormente, habrá que comprobar que el diseño puede ser implementado adecuadamente en una FPGA y que ésta responde como se espera. Para ello se deben diseñar una serie de pruebas a nivel Hardware que permitan identificar posibles fallos del diseño.

Para dar soporte a ambos tipos de prueba se han diseñado una serie de componentes que permiten simular el comportamiento de los analizadores ante un sistema real. Concretamente,

se han diseñado un modelo de simulación para pruebas de dispositivos maestro SPI y otro de dispositivos esclavos para poder comprobar si el analizador SPI captura las tramas transferidas entre ellos. Además, se ha empleado un modelo de simulación de dispositivos maestro I2C y otro de esclavos ya diseñados para trabajos previos y que han sido verificados, por lo que permitirá comprobar que el analizador I2C realiza su trabajo de forma adecuada ante un sistema real.

En primer lugar, se describirán las características de diseño de estos simuladores, presentándose su interfaz de puertos así como la funcionalidad que posee cada uno y la implementación que los compone.

Posteriormente se definirán las diversas pruebas realizadas, tanto a nivel de simulación como a nivel de hardware, y se presentarán los resultados obtenidos en cada una de ellas.

4.1 Diseño de modelos de simulación para pruebas

Para facilitar la comprensión y posterior análisis de cada uno de los protocolos expuestos y permitir la verificación del diseño realizado, se han diseñado varios bloques que modelan el comportamiento de un dispositivo, tanto maestro como esclavo, que se conectaría a uno de los buses analizados. Estos módulos también han sido diseñados empleando el lenguaje de descripción de hardware VDHL de modo que pueden incluirse en el mismo proyecto donde se ha implementado el analizador. Esto permite realizar pruebas, tanto a nivel de simulación como a nivel hardware, empleando un solo proyecto y una única FPGA de manera que se reduce el número de recursos a emplear en la verificación. Los modelos de simulación implementados dan lugar a un circuito semejante al de los dispositivos que simula por lo que se puede asumir que las pruebas se realizarán ante un sistema real.

En primer lugar se describirán los bloques implementados para el analizador SPI, realizándose una pequeña descripción de ellos ya que no es el objetivo fundamental de este proyecto. A continuación se presentarán los bloques empleados para la verificación del analizador I2C.

4.1.1 SPI Master

Este bloque es un modelo de simulación que tiene el mismo comportamiento que un dispositivo maestro conectado a un bus SPI. Posee una serie de puertos que permiten su interacción con otros dispositivos y que componen la interfaz mostrada en la siguiente imagen:

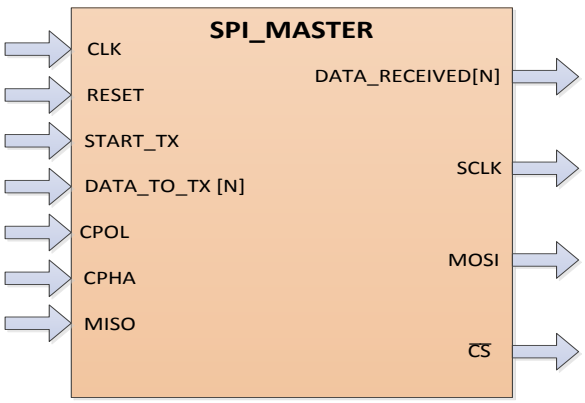


Figura 43: Interfaz del modelo de simulación maestro SPI

Interfaz del diseño (entradas/salidas)

Las tablas expuestas a continuación, recogen el conjunto de parámetros genéricos que permiten configurar el dispositivo y la interfaz que éste presenta:

- Genéricos:

Genérico	Tipo	Descripción
G_CLK_PERIOD	Natural	Parámetro que define el periodo del reloj del sistema expresado en picosegundos. Se emplea para configurar la velocidad de transmisión en baudios (bits/s)
G_SCLK_PERIOD	Real	Periodo deseado para el reloj de sincronización SCLK del bus SPI expresado en picosegundos. Se emplea para configurar la velocidad de transmisión en baudios (bits/s)
G_NUMBER_OF_BITS	Natural	Número de bits que contendrá cada mensaje. Este parámetro permite definir la longitud del mensaje SPI

Tabla 29: Genéricos del modelo de simulación de maestro SPI

Mediante la asignación de valores al parámetro G_SCLK_PERIOD se puede obtener la velocidad de transmisión deseada ya que ambos parámetros se relacionan como muestra la siguiente fórmula:

$$velocidad\ de\ transmisión\ \left(\frac{bits}{s}\right) = \frac{1}{1/G\_SCLK\_PERIOD} = G\_SCLK\_PERIOD$$

Ecuación 3: Velocidad de transmisión del modelo de simulación de maestro SPI

- Puertos

Puerto	Tipo	Número de bits	Nivel activo	Valor de Reset	Descripción
CLK	Entrada	1	Flanco de subida		Entrada del reloj empleada para sincronización del sistema
RESET	Entrada	1	‘1’		Terminal de reinicio del sistema
MISO	Entrada	1	No aplica		Entrada de datos del maestro provenientes del esclavo
CPOL	Entrada	1	No aplica		Entrada de polaridad del reloj SCLK

Puerto	Tipo	Número de bits	Nivel activo	Valor de Reset	Descripción
CPHA	Entrada	1	No aplica		Entrada que define la configuración de fase del modo SPI
START_TX	Entrada	1	'1'		Puerto para solicitar el envío de un nuevo mensaje
DATA_TO_TX	Entrada	Depende de G_NUMBER_OF_BITS	No aplica		Entrada que contiene el mensaje a enviar
MOSI	Salida	1		Alta impedancia	Salida de datos del maestro hacía el esclavo
SCLK	Salida	1		Depende de CPOL	Línea de reloj serie del bus SPI
CS	Salida	8		'1'	Selector de esclavo
DATA_RECEIVED	Salida	Depende de G_NUMBER_OF_BITS		Todos los bits a '0'	Mensaje recibido del esclavo

Tabla 30: Interfaz del modelo de simulación de maestro SPI

Funcionalidad:

Este bloque simula el comportamiento de un maestro del bus SPI por lo se encarga de gestionar las transmisiones que se efectúan a través del bus. Cuando se solicita una nueva transferencia a través de su entrada START_TX, el sistema selecciona al esclavo mediante la fijación de la salida CS a nivel bajo. Cuando ha transcurrido la mitad del periodo de envío de un bit (que depende de la velocidad de transmisión fijada) el sistema activa el reloj de sincronización SCLK, cuyo periodo depende de la velocidad de transmisión seleccionada como se ha explicado previamente. En caso de que la entrada CPHA esté a nivel bajo, el sistema enviará a través de su salida MOSI el bit más significativo del mensaje, contenido en su entrada DATA_TO_TX, entre la bajada de CS y el segundo flanco de reloj como indican los modos SPI 1 y 2 (ver sección 2.1.3). El envío del resto de bits se hará entre los flancos pares del reloj como indican dichos modos hasta enviar el LSB del mensaje. Si la entrada CPHA está a nivel alto, el sistema esperará a activar el reloj de sincronización SCLK para comenzar la transmisión, dejando su salida MOSI en alta impedancia hasta ese momento. El MSB del mensaje se enviará cuando se produzca la activación del reloj y el siguiente bit se enviará durante el segundo ciclo

del reloj SCLK de modo que cada bit se enviará en un ciclo de SCLK. El último bit del mensaje permanecerá en la línea MOSI durante la mitad de un periodo de SCLK tras la desactivación de éste ya que el dispositivo esclavo leerá el dato en ese momento como indican los modos 3 y 4 del protocolo SPI (ver sección 2.1.3). De esta manera se consigue que el dato permanezca estable en la línea MOSI el tiempo suficiente para que sea leído por el esclavo. Finalizada la transmisión, el dispositivo pondrá su salida MOSI en alta impedancia para evitar que los esclavos interpreten nuevos datos.

Además, este dispositivo es capaz de recibir datos de un esclavo a través de su entrada MISO. La recepción de cada bit dependerá del modo SPI seleccionado, leyéndose en el flanco adecuado del reloj SCLK como indica la definición del protocolo (ver sección 2.1.3).

Implementación:

A grandes rasgos, este módulo incluye los siguientes elementos:

- Un contador encargado de gestionar la generación del reloj de sincronización SCLK a la frecuencia debida.
- Un contador de flancos que permite calcular el número de bits transferidos.
- Un registro de desplazamiento, cuyo tamaño depende de la longitud de mensaje configurada, empleado para almacenar los datos enviados y recibidos. El MSB de este registro se conecta a la salida MOSI para enviar el bit adecuado y el LSB se conecta a la entrada MISO para recibir los datos de manera que un único registro gestiona ambas operaciones.
- Un buffer triestado, controlado desde la máquina de estados, se sitúa en la salida MOSI para permitir a otros dispositivos escribir en el bus.
- El modulo incluye una máquina de estados empleada para gestionar el envío de mensajes, habilitando la generación del reloj SCLK y gestionando el envío y recepción de datos en función del modo SPI seleccionado.

4.1.2 SPI Slave

Este componente se encarga de ejercer el comportamiento de un esclavo conectado a un bus SPI. El diseño posee una interfaz que aparece reflejada en la imagen expuesta a continuación y que será descrita posteriormente. Además, se realizará una breve descripción de la funcionalidad del módulo y los recursos principales empleados para el diseño.

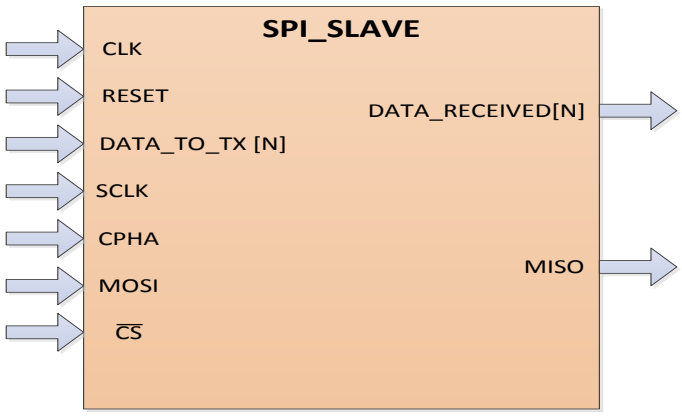


Figura 44: Interfaz del simulador de esclavo SPI

Interfaz del diseño (entradas/salidas)

Las tablas expuestas a continuación, recogen el conjunto de parámetros genéricos que permiten configurar el dispositivo y la interfaz que éste presenta:

- Genéricos:

Genérico	Tipo	Descripción
G_CLK_PERIOD	Natural	Parámetro que define el periodo del reloj del sistema expresado en picosegundos. Se emplea para configurar la velocidad de transmisión en baudios (bits/s)
G_SCLK_PERIOD	Real	Periodo deseado para el reloj de sincronización SCLK del bus SPI expresado en picosegundos. Se emplea para configurar la velocidad de transmisión en baudios (bits/s) al igual que en el dispositivo maestro.
G_NUMBER_OF_BITS	Natural	Número de bits que contendrá cada mensaje. Este parámetro permite definir la longitud del mensaje SPI

Tabla 31: Genéricos del modelo de simulación de esclavo SPI

- Puertos

Puerto	Tipo	Número de bits	Nivel activo	Valor de Reset	Descripción
CLK	Entrada	1	Flanco de subida		Entrada de reloj del sistema
RESET	Entrada	1	‘1’		Puerto de reinicio del componente
CPHA	Entrada	1	No aplica		Entrada que define la configuración de fase del modo SPI
MOSI	Entrada	1	No aplica		Entrada de datos del esclavo proveniente del maestro
SCLK	Entrada	1	No aplica		Entrada de reloj de sincronización del bus SPI

Puerto	Tipo	Número de bits	Nivel activo	Valor de Reset	Descripción
CS	entrada	1	'0'		Entrada de selección de chip del esclavo
DATA_TO_TX	Entrada	Depende de G_NUMBER_OF_BITS	No aplica		Entrada que contiene el mensaje a enviar
MISO	Salida	1		Alta impedancia	Salida de datos del esclavo hacía el maestro
DATA_RECEIVED	Salida	Depende de G_NUMBER_OF_BITS		Todos los bits a '0'	Mensaje recibido del maestro

Tabla 32: Interfaz del modelo de simulación de esclavo SPI

Funcionalidad:

Este módulo se encarga de recibir los mensajes enviados por el maestro SPI a través de su entrada MOSI y enviar los mensajes que recibe por su entrada DATA_TO_TX al maestro a través de su salida MISO. Cuando el dispositivo detecta que ha sido seleccionado para una transmisión a través de la identificación del nivel bajo en su entrada CS, el dispositivo evalúa el modo de transmisión a realizar mediante su entrada CPHA ya que la polaridad la identifica a través de la línea de reloj SCLK. Si la entrada está a nivel bajo, enviará sus datos y recibirá los transferidos por el maestro según los modos SPI 1 o 2, dependiendo de la polaridad del reloj SCLK (ver sección 2.1.3). Por el contrario, si la entrada CPHA está a nivel alto, las transmisiones se efectuarán según los modos 3 o 4 del protocolo SPI.

Cuando la transmisión haya finalizado, el dispositivo esclavo pondrá su salida MISO en alta impedancia para permitir que otros esclavos escriban en el bus.

Implementación:

Principalmente este módulo emplea los siguientes recursos:

- Un contador de flancos permite calcular el número de bits que han sido transmitidos.
- Un contador auxiliar se emplea para conseguir que el dispositivo esclavo mantenga el último bit del mensaje en la línea MISO durante un tiempo aproximado de medio ciclo de SCLK si se emplean los modos 3 o 4 de transmisión. De esta manera se asegura que el dato es estable cuando el maestro lo lea tras el último flanco del reloj SCLK.
- Un registro d desplazamiento, de tamaño variable en función de la longitud del mensaje, se emplea para almacenar los datos a enviar y los datos a recibir de forma semejante a la usada en el simulador de dispositivo maestro.
- Un buffer triestado se sitúa en la salida MISO para permitir que otros esclavos escriban en el bus.

- Por último, una FSM se encarga de gestionar la transmisión manejando todos los componentes anteriormente descritos para conseguir que las transmisiones se realicen según el modo SPI deseado.

4.1.3 Control de Master y Slave SPI

Este bloque se ha diseñado como sistema de control de los modelos de simulación maestro y esclavo diseñados para el analizador SPI. Dado que el sistema se implementará en una FPGA donde no se dispondrá de capacidad de control de los dispositivos, este componente se encargará de facilitar los datos a enviar a cada uno de los dispositivos y controlar el dispositivo maestro para que inicie las transmisiones. Además, este bloque incluye al analizador SPI de modo que permite contener en un único proyecto a los sistemas conectados al bus (maestro y esclavo) y al analizador para que el conjunto pueda ser implementado en una única FPGA.

La imagen mostrada a continuación, refleja la interfaz de dicho componente y las conexiones presentes entre los bloques que incluye.

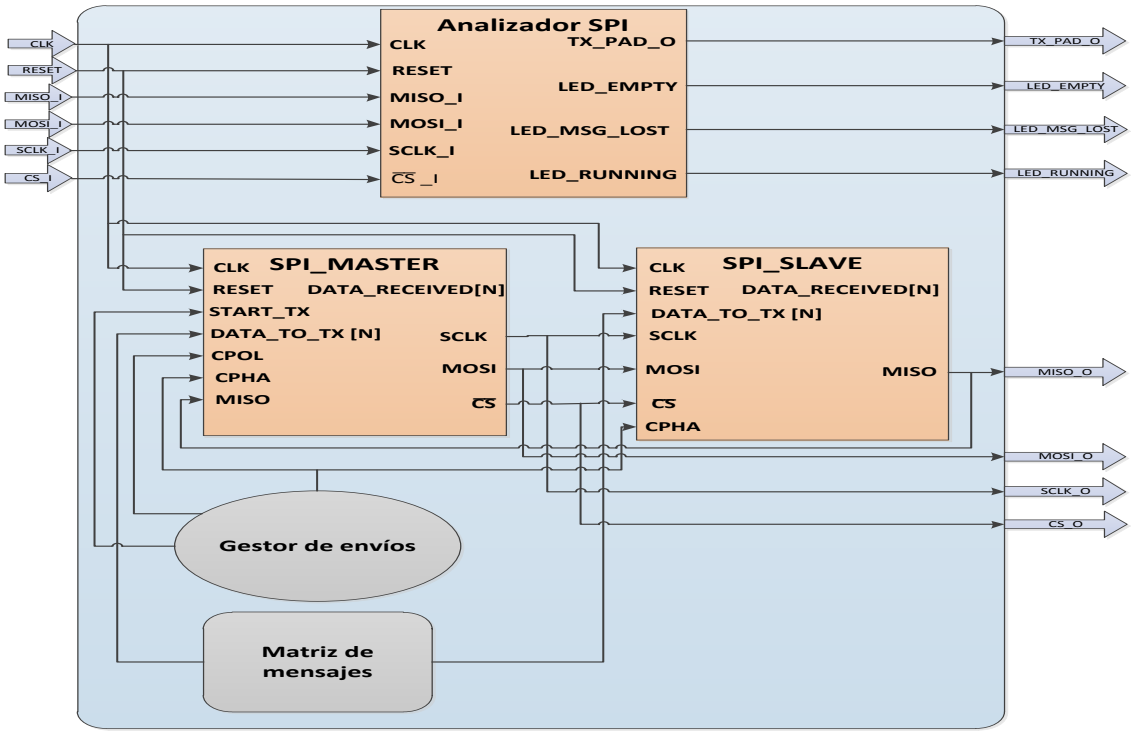


Figura 45: Diagrama de bloques del control de maestro y esclavo SPI

A continuación, se describirán en una tabla los puertos que presenta este componente y se definirá brevemente cuál es su funcionalidad e implementación.

Puerto	Tipo	Número de bits	Nivel activo	Valor de Reset	Descripción
CLK	Entrada	1	Flanco de subida		Entrada del reloj del sistema

Puerto	Tipo	Número de bits	Nivel activo	Valor de Reset	Descripción
RESET	Entrada	1	‘1’		Puerto empleado para reiniciar el sistema
MISO_I	Entrada	1	‘1’		Conexión a la línea MISO del bus SPI para el analizador.
MOSI_I	Entrada	1	‘1’		Conexión a la línea MOSI del protocolo SPI para el analizador.
SCLK_I	Entrada	1	‘1’		Puerto de recepción del reloj de SPI para el analizador.
CS_I	Entrada	1	‘0’		Conexión de la línea de selección de esclavo del bus SPI para el analizador.
LED_EMPTY	Salida	1		‘1’	Puerto que indica al exterior que el analizador SPI no tiene mensajes por enviar
LED_MSG_LOST	Salida	1		‘0’	Salida empleada para mostrar perdidas de mensajes en la captura
LED_RUNNING	Salida	1		‘0’	Terminal conectado a un LED que indica que el analizador está capturando un mensaje
TX_PAD_O	Salida	1		‘1’	Puerto empleado para la transmisión serie al ordenador de los mensajes capturados.
MISO_O	Salida	1		‘1’	Puerto de salida de datos del simulador de dispositivo maestro SPI
MOSI_O	Salida	1		‘1’	Puerto de salida de datos del simulador de dispositivo esclavo SPI

Puerto	Tipo	Número de bits	Nivel activo	Valor de Reset	Descripción
SCLK_O	Salida	1		Depende de la polaridad seleccionada	Puerto de salida del reloj de sincronización del simulador de maestro del bus SPI
CS_O	Salida	1		'1'	Puerto de salida de selección de esclavo del simulador de maestro del bus SPI

Figura 46: Interfaz del bloque de control del maestro y el esclavo SPI

Funcionalidad:

Este bloque ha sido empleado para permitir conectar el analizador de protocolos SPI a los modelos de simulación de los dispositivos esclavo y maestro SPI y gestionar las transmisiones que se realizan entre estos. El sistema permite controlar, de manera totalmente automática, el dispositivo maestro del bus SPI. Cuando el sistema sale del estado de reinicio, este bloque envía al maestro y al esclavo los datos que se desean transferir en cada caso. Mediante la activación de la entrada START_TX del maestro, el dispositivo maestro comienza la transmisión, haciendo que se active también el dispositivo esclavo al ser seleccionado y que se transfieran datos entre ellos. Un contador de mensajes enviados se encarga de gestionar el modo de envío del protocolo SPI que se empleará en cada caso de modo que asigna diferentes valores a los parámetros CPHA y CPOL para conseguir transferencias en distintos modos. Cuando el maestro ha terminado una transmisión, este módulo comprueba el valor del contador de mensajes, cuyo límite coincide con el número de mensajes que han sido configurados, para enviar otro mensaje o parar el sistema cuando todos han sido enviados.

De esta manera es posible realizar pruebas a nivel de hardware del analizador sin necesidad de controlar de manera externa los modelos de simulación SPI, pudiendo comprobarse el funcionamiento ante los diversos modos de transmisión SPI.

Implementación:

La implementación de este módulo es bastante simple ya que únicamente contiene:

- Un contador de mensajes cuyo tamaño es configurable para permitir enviar mayor o menor número de mensajes.
- Una matriz de datos que contiene los mensajes que se desean enviar. El tamaño de esta matriz es configurable de modo que el ancho de cada elemento define la longitud en bits de los mensajes a enviar. La profundidad o largo de la matriz define el número de mensajes que se desean enviar en la prueba a realizar.
- Un proceso encargado de gestionar el comienzo de las transferencias y la configuración del modo de transferencia SPI a emplear. Mediante la evaluación del valor del contador de mensajes, se asignan diferentes valores a CPHA y CPOL para que cada mensaje se envíe según el modo 1, 2, 3 o 4 de transferencia SPI (ver sección 2.1.3).

4.1.4 I2C Master

Este bloque se encarga de simular el comportamiento de un dispositivo maestro conectado a un bus I2C y, por tanto, se encarga de gestionar las transmisiones y definir el tipo de operación a realizar en cada uno de ellos. El módulo fue diseñado para un trabajo previo y ha pasado numerosas pruebas, tanto a nivel de simulación como a nivel físico implementándose en diversas FPGAs, por lo que compone un elemento de verificación fiable. Este bloque fue diseñado por el autor de este proyecto en colaboración con José Antonio Fernández de Blas, ingeniero de desarrollo Hardware en la compañía Indra Sistemas S.A. En este apartado se definirá la interfaz que presenta este elemento así como se describirá brevemente la funcionalidad que presenta y la implementación que incluye. La imagen mostrada a continuación contiene la interfaz del módulo que será descrita a continuación:

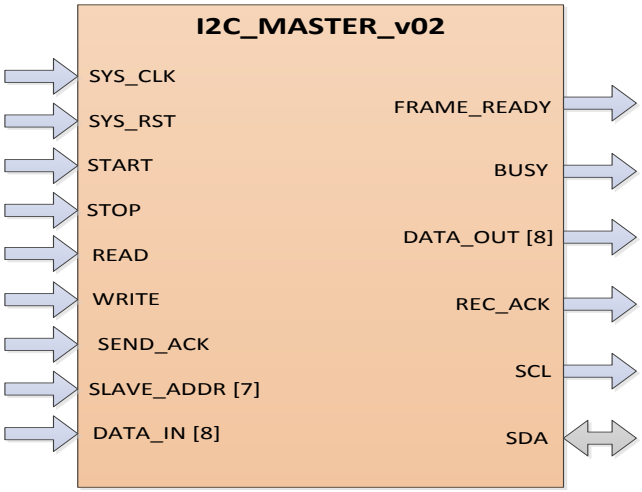


Figura 47: Interfaz del modelo de simulación de maestro I2C

Interfaz del diseño (entradas/salidas)

Las tablas expuestas a continuación, recogen el conjunto de parámetros genéricos que permiten configurar el dispositivo y la interfaz que éste presenta:

- Genéricos:

Genérico	Tipo	Descripción
CLK_FREQ	Natural	Parámetro que define el periodo del reloj del sistema expresado en picosegundos. Se emplea para configurar la velocidad de transmisión en baudios (bits/s)
BAUD	Real	Periodo deseado para el reloj de sincronización SCLK del bus SPI expresado en picosegundos. Se emplea para configurar la velocidad de transmisión en baudios (bits/s) en el bus I2C.

Tabla 33: Genéricos del modelo de simulación de maestro I2C

El parámetro BAUD permite definir la velocidad de transferencia que empleará el bus I2C ya que ya que ésta se relaciona con él como define la siguiente fórmula:

$$velocidad\ de\ transmisión\ \left(\frac{bits}{s}\right) = \frac{1}{1/BAUD} = BAUD$$

Ecuación 4: Velocidad de transmisión del modelo de simulación de maestro I2C

- Puertos

Puerto	Tipo	Número de bits	Nivel activo	Valor de Reset	Descripción
SYS_CLK	Entrada	1	Flanco de subida		Entrada del reloj de sistema
SYS_RESET	Entrada	1	'1'		Terminal de reinicio del módulo
START	Entrada	1	'1'		Entrada para solicitar el inicio de una transmisión al maestro I2C
STOP	Entrada	1	'1'		Entrada para solicitar el fin de una transmisión al maestro I2C
READ	Entrada	1	'1'		Puerto para indicar que se desea realizar una operación de lectura (del esclavo)
WRITE	Entrada	1	'1'		Entrada que indica que se desea realizar una operación de escritura (al esclavo)
SEND_ACK	Entrada	1	'1'		Puerto para indicar el envío de una señal de asentimiento a un byte recibido
SLAVE_ADDR	Entrada	7	No aplica		Puerto que contiene la dirección del esclavo con el que se desea comunicar
DATA_IN	Entrada	8	No aplica		Puerto que contiene el byte a enviar en cada momento

Puerto	Tipo	Número de bits	Nivel activo	Valor de Reset	Descripción
FRAME_READY	Salida	1		‘0’	Salida para solicitar un nuevo dato a enviar
BUSY	Salida	1		‘0’	Salida que indica que el dispositivo está ocupado
REC_ACK	Salida	1		‘0’	Puerto que indica si el esclavo ha enviado una señal de asentimiento a un byte enviado
DATA_OUT	Salida	8		Todos los bits a ‘0’	Byte recibido del esclavo
SCL	salida	1		‘1’	Salida del reloj de sincronización del bus I2C
SDA	Entrada/salida	1	‘0’	Alta impedancia	Puerto de envío/recepción de datos del protocolo I2C

Tabla 34: Interfaz del modelo de simulación de maestro I2C

Funcionalidad:

Este módulo se encarga de controlar las transferencias realizadas a través del bus I2C. Cuando su entrada START está a nivel alto, el sistema comienza la transferencia mediante la puesta de la línea SDA a nivel bajo, creando la condición de inicio definida previamente (ver sección 2.2.3). A continuación, el sistema activa el reloj de sincronización SCL, el cual trabajará a la frecuencia definida por la velocidad de transferencia. En ese momento comienza la transmisión de la dirección del esclavo, que se realiza de manera secuencial comenzando por el MSB de ésta para terminar con su LSB. Cada bit, es puesto en la línea SDA durante el tiempo a nivel bajo del reloj SCL para asegurar que el dato es estable durante el tiempo en activo de éste como marca la especificación del protocolo ([REF-2] Especificación del bus I2C). Finalizado el envío de los 7 bits de la dirección del esclavo, el sistema activa su salida FRAME_READY para solicitar el tipo de operación a realizar, de lectura o de escritura. Cuando el sistema detecta un nivel alto en su entrada de READ o WRITE, envía el bit de operación, siendo éste un 0 cuando la operación es de escritura o un 1 cuando se trata de una lectura. Posteriormente, en función del tipo de operación seleccionado el sistema transferirá un byte o lo recibirá desde el esclavo comenzando por el MSB y terminando con la transmisión del LSB. Cada vez que se envíe o reciba un byte completo, el sistema activará su salida FRAME_READY para saber qué debe hacer a continuación, pudiendo escribir/recibir otro byte o finalizar la transmisión mediante su entrada de STOP. Además, el sistema informará al bloque que gestiona su control de la recepción de la señal de asentimiento de cada byte mediante su

salida REC_ACK. De forma análoga, cuando el sistema recibe un byte debe ponerse a nivel alto su entrada SEND_ACK para que envíe la señal de asentimiento de recepción al esclavo.

Implementación:

Este módulo está compuesto principalmente por:

- Un contador, cuyo tamaño se define en función de la velocidad de transmisión requerida, empleado para generar el reloj de sincronización SCL.
- Un contador de 3 bits empleado para controlar el número de bits enviados/recibidos durante la transmisión de un byte.
- Dos registros de 8 bits empleados para enviar y recibir los datos respectivamente. El sistema utiliza el contador de bits para seleccionar el bit del registro que debe enviar en cada momento.
- Una máquina de estados que controla el conjunto del sistema. Dicha FSM contiene numerosos estados que permiten el inicio de las transmisiones y controlan el envío y recepción de los datos.

4.1.5 I2C Slave

Este módulo ha sido diseñado para simular el comportamiento de un dispositivo esclavo conectado a un bus I2C. El diseño contiene un conjunto de constantes que definen la dirección de varios esclavos, de modo que puede actuar como si se tratase de varios esclavos distintos conectados al mismo bus. Como en el caso del maestro, este bloque fue diseñado en trabajos previos por el autor de este proyecto y el ingeniero de desarrollo hardware José Antonio Fernández de Blas y, por tanto, ha pasado numerosas pruebas por lo que podrá ser utilizado para corroborar el funcionamiento del analizador I2C. La interfaz que incluye este bloque se puede observar en la siguiente imagen y será descrita a continuación de ésta. Además en este apartado se describirá, de forma breve, la funcionalidad de dicho elemento y la implementación que incluye.

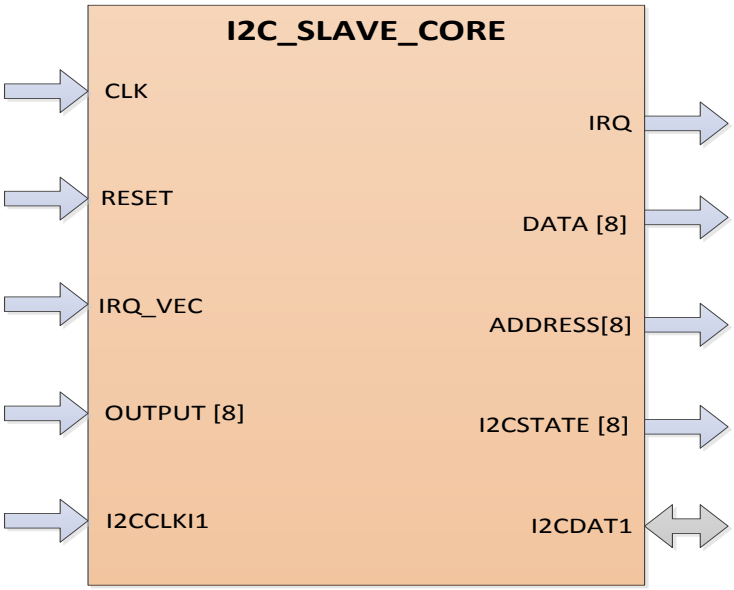


Figura 48: Interfaz del modelo de simulación de esclavo I2C

Interfaz del diseño (entradas/salidas):

Puerto	Tipo	Número de bits	Nivel activo	Valor de Reset	Descripción
CLK	Entrada	1	Flanco de subida		Entrada del reloj general de sistema
RESET	Entrada	1	'1'		Terminal de reinicio del módulo
IRQ_VEC	Entrada	1	'1'		Entrada que indica que el controlar está manejando una interrupción. Se usa para impedir la recepción de nuevos datos por estar ocupado el control
I2CCLKI1	Entrada	1	Flanco de subida		Entrada de recepción del reloj de sincronización del bus I2C.
OUTPUT	Entrada	8	No aplica		Puerto que contiene el byte que se desea enviar en cada momento
IRQ	Salida	1		'0'	Salida para generar una interrupción en el sistema de control
DATA	Salida	8		Todos los bits a '0'	Byte recibido del maestro a través de la línea SDA
ADDRESS	Salida	8		Todos los bits a '0'	Salida que contiene la dirección del esclavo que ha sido seleccionado para la transmisión.
I2CSTATE	Salida	8		Todos los bits a '0'	Salida que contiene un vector de bits que identifica el estado en el que se encuentra el sistema para que lo identifique el control y controles la transmisión
SDA	Entrada/salida	1	'0'	Alta impedancia	Puerto de envío/recepción serie de datos del protocolo I2C

Tabla 35: Interfaz del modelo de simulación de esclavo I2C

Funcionalidad:

Este bloque se encarga de simular el comportamiento de varios dispositivos esclavos conectados al bus I2C gracias a que contiene un conjunto de constantes que definen la dirección de cada esclavo que se desea simular. Cuando se detecta una condición de inicio en el bus, el dispositivo almacena cada uno de los bits de la dirección cuando se detecta la subida del reloj SCL como indica la especificación del bus (ver [REF-2] Especificación del bus I2C). Una vez recibida la dirección completa y el bit de lectura/escritura, el sistema evalúa si la dirección recibida coincide con la asignada a alguno de los esclavos simulados o se trata de una llamada general (dirección = "0000000") para enviar la señal de asentimiento de direccionamiento al esclavo. En ese momento, el módulo envía a su salida ADDRESS los 7 bits de dirección y el tipo de operación a realizar para que el control defina el siguiente paso que se debe llevar a cabo. Si la operación es de lectura, el dispositivo capturará cada bit cuando se detecte la subida del reloj SCL y enviará la señal de asentimiento de recepción durante el noveno ciclo de la transmisión del byte. Si la operación es de escritura, el dispositivo enviará primero el MSB del byte contenido en su entrada OUTPUT y finalizará la transmisión con el envío del LSB de dicho byte. El cambio en la línea SDA se producirá tras la bajada de SCL para asegurar que el dato es estable cuando el reloj vuelva a subir. Finalizado el envío del LSB, el dispositivo esperará la señal de asentimiento del maestro para pasar a enviar un nuevo byte o esperar la llegada de una condición de parada para volver al reposo.

Implementación:

- Un registro doble se sitúa después de cada entrada para evitar problemas de metaestabilidad y adaptar las señales al dominio del reloj del sistema.
- Un contador de 3 bits es empleado para calcular el número de bits que han sido enviados o recibidos durante la transferencia de un byte.
- Un conjunto de constantes de 7 bits contiene la dirección asignada a cada uno de los esclavos simulados en este bloque, pudiendo simularse hasta 12 esclavos de manera simultánea.
- Un registro de 8 bits es empleado para almacenar el dato que se desea transferir en las operaciones de lectura. Otros registros similares son empleados para almacenar el byte recibido y la dirección del esclavo seleccionado.
- Por último, una máquina de estados se encarga de controlar el funcionamiento de este bloque. Esta FSM controla la recepción de la dirección del esclavo, la lectura de los bytes recibidos y el envío de los bytes transmitidos. Además se encarga de enviar y evaluar la recepción de las señales de asentimiento correspondientes a cada byte.

4.1.6 Control de Master y Slave I2C

El bloque presentado en este apartado ha sido diseñado para controlar, de forma automática, los modelos de simulación para dispositivos maestros y esclavos del bus I2C. Dado que los modelos de simulación presentados previamente han sido diseñados para controlarse a través de un microprocesador, ha sido necesario diseñar un elemento capaz de controlar estos diseños desde dentro de la propia FPGA. La solución implementada contiene a los dispositivos maestros y esclavos del bus y al analizador I2C para que capture los mensajes transmitidos entre ellos. Este diseño no emplea algunas de las salidas de los simuladores ya que no son

necesarias para este caso y presenta una interfaz para comunicar con el exterior muy simple que puede observarse en la imagen expuesta a continuación:

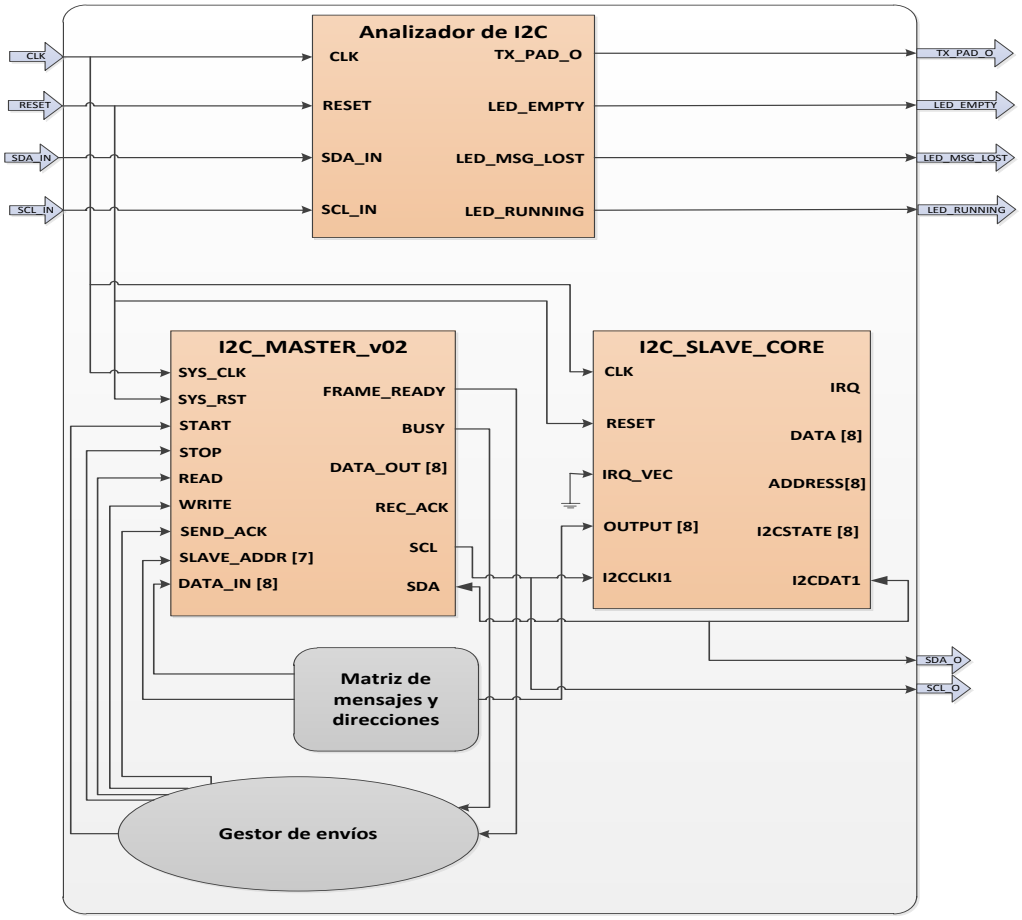


Figura 49: Diagrama de bloques del control de modelos de simulación I2C

Interfaz del diseño (entradas/salidas):

Puerto	Tipo	Número de bits	Nivel activo	Valor de Reset	Descripción
CLK	Entrada	1	Flanco de subida		Entrada del reloj general del sistema
RESET	Entrada	1	'1'		Puerto empleado para reiniciar el sistema
SDA_IN	Entrada	1	No aplica		Conexión a la línea de datos serie del bus I2C
SCL_IN	Entrada	1	No aplica		Puerto de recepción del reloj serie del bus I2C

Puerto	Tipo	Número de bits	Nivel activo	Valor de Reset	Descripción
LED_EMPTY	Salida	1		‘1’	Puerto que indica al exterior que el analizador SPI no tiene mensajes por enviar
LED_MSG_LOST	Salida	1		‘0’	Salida empleada para mostrar perdidas de mensajes en la captura
LED_RUNNING	Salida	1		‘0’	Terminal conectado a un LED que indica que el analizador está capturando un mensaje
TX_PAD	Salida	1		‘1’	Puerto empleado para la transmisión serie al ordenador de los mensajes capturados.
SDA_O	Salida	1		‘1’	Salida de datos de los módulos de simulación
SCL_O	Salida	1		‘1’	Salida del reloj de sincronización empleado pos los módulos de simulación

Tabla 36: Interfaz del bloque de control de los modelos de simulación I2C

Funcionalidad:

Es módulo se encarga de controlar los modelos de simulación de dispositivos maestro y esclavos para que el analizador pueda capturar los datos transferidos entre ellos. El sistema se encarga, cuando no se encuentra en estado de reinicio, de controlar el dispositivo maestro para que efectúe las transmisiones necesarias y facilita los datos que debe escribir el esclavo en el bus cuando la operación es de lectura.

En primer lugar, si el dispositivo detecta que el maestro no está ocupado y hay algún mensaje que enviar, comanda al maestro para que inicie una transferencia. Cuando el maestro lo solicita a través de su salida FRAME_READY, el módulo envía a éste la dirección del esclavo con el que se desea comunicar. Dicha dirección está contenida en una matriz de datos que incluye todas las direcciones de esclavo que presenta el simulador de esclavo. De esta forma se simula el direccionamiento de distintos esclavos presentes en el bus.

A continuación, se le indica al maestro el tipo de operación que se desea realizar activando su entrada correspondiente (READ o WRITE). El diseño direccionará a cada esclavo en dos ocasiones, una para escribir datos y otra para leerlos de éste.

Si la operación es de escritura, el módulo cargará el byte a enviar al maestro y controlará a este de forma adecuada para que lo transmita. Por el contrario, si la operación es de lectura el byte a enviar será facilitado al esclavo para que este realice la transmisión.

Mediante un contador el sistema controla el envío de cada mensaje, de manera que el primer mensaje contendrá un solo byte, el segundo dos y así sucesivamente hasta alcanzar el máximo número de bytes y mensajes fijados.

De esta manera, se simulan transferencias, tanto de escritura como de lectura, con diferentes longitudes (en bytes) del mensaje. Gracias a esto se puede comprobar el funcionamiento del analizador ante diferentes condiciones de transmisión.

Implementación:

A grandes rasgos, este módulo incluye los siguientes elementos:

- Un contador que controla el número de mensajes transmitidos para transferir los deseados.
- Un contador auxiliar se encarga de controlar el número de bytes que contendrá cada mensaje. Con el comienzo de un mensaje, este contador se pone a cero y su valor aumenta con el envío de cada byte hasta alcanzar el valor del contador de mensajes. Esto permite que el primer mensaje contenga un solo byte, el segundo dos y así sucesivamente hasta el límite fijado.
- Una matriz de ancho de 8 bits, para almacenar los bytes a enviar, y profundidad variable, para cambiar el número de mensajes a transmitir, se emplea para almacenar los datos a enviar. Estos datos se usarán tanto como direcciones de esclavo a direccionar como datos a transmitir, reduciendo el consumo de recursos empleados en la implementación. Para direccionar al esclavo, se emplearán los 7 bits más significativos de cada byte almacenado.
- Por último, una máquina de estados se encarga de controlar al dispositivo maestro y facilitar los datos a transmitir por el esclavo en las operaciones de lectura. En función del contador de mensajes enviados y el contador de bytes contenidos en cada mensaje, la FSM hará que se transmitan los mensajes con diferentes longitudes. Además, una señal se emplea para controlar que cada mensaje sea transmitido desde el maestro al esclavo (escritura) o viceversa (lectura).

4.2 Pruebas de simulación

Una vez diseñado cada uno de los módulos anteriormente expuestos, es necesario corroborar su correcto funcionamiento. Para ello, el primer paso es realizar diversas pruebas de simulación que permitan verificar el comportamiento adecuado del diseño ante diversos estímulos.

Para realizar dichas pruebas se ha empleado el simulador proporcionado con el entorno Xilinx ISE 13.4, entorno sobre el que se hablará en el apartado de pruebas a nivel Hardware. Este simulador recibe el nombre de ISim y su versión incluida en el entorno de desarrollo es la 0.87xd.

Existen diversos simuladores con capacidad de simular diseños realizados en VHDL, siendo muy conocido el entorno desarrollado por Mentor Graphics Corporation que recibe el nombre de ModelSim.

Sin embargo, en este caso se ha elegido ISim ya que está integrado en el entorno de desarrollo Xilinx ISE 13.4 y permite crear módulos de banco de prueba para cada uno de los bloques

diseñados de forma automática. Seleccionando la entidad que se desea simular, el programa crea el banco de prueba y las señales básicas para poder crear los estímulos necesarios para éste. Además, el simulador incluye las librerías de componentes de propiedad intelectual (*IP Cores*) que permiten simular el comportamiento de éstos en caso de que hayan sido empleados en el diseño, lo cual permite depurar el funcionamiento del conjunto de manera sencilla.

Una vez definido el entorno empleado, se describirán las pruebas realizadas a cada uno de los módulos desarrollados en el proyecto. En primer lugar, se describirán las pruebas realizadas a cada uno de los módulos desarrollados. Posteriormente, se detallarán las pruebas realizadas al conjunto de bloques que compone cada analizador:

Pruebas a nivel de bloque:

- SPI Sampler

Un banco de pruebas de simulación específicamente diseñado para este bloque, y que contiene a los simuladores de maestro y esclavo SPI, comprueba que:

  - Se identifica el comienzo de una transmisión a través de la línea de selección de esclavo (CS).
  - Se capturan el valor de las entradas MOSI y MISO en cada flanco del reloj SCLK.
  - Se identifica el valor de la configuración de fase de la comunicación.
  - El contador de flancos funciona adecuadamente.
  - Se capturan los bits enviados por las líneas MOSI y MISO según marca el modo de transferencia SPI.
  - El número de bits del mensaje calculados coincide con la longitud de éste y se refleja en la salida NUM_BITS.
  - La polaridad del reloj SCLK se identifica adecuadamente.
  - El mensaje capturado a través de las líneas MOSI y MISO coincide con el transferido y se muestra en las salidas DATA_MOSI y DATA_MISO respectivamente.
  - Se identifica el final de una transmisión mediante la subida a nivel alto de la línea CS. Esta comprobación también permite saber que se detectarán fallos en la longitud del mensaje si la transmisión es más corta de lo esperado.
  - La salida que indica funcionamiento (RUNNING) se mantiene activa mientras la entrada CS está a nivel bajo, lo que indica que se está capturando. Esto permite validar que se podrá identificar que la transmisión se ha cortado sin finalizar.
  - La salida NEW_MSG se pone a nivel alto cuando finaliza la captura de un mensaje.

- Bloque de control SPI

Un banco de pruebas de simulación valida que el componente:

  - Es capaz de almacenar los datos del mensaje en la memoria de 256 bits cuando es requerido.
  - Almacena los datos de longitud, polaridad y fase de manera adecuada en la memoria de 16 bits.
  - Comienza la transferencia de un mensaje cuando las memorias contienen datos y permanece en reposo cuando no.

- Envía en cada momento los datos adecuados a los conversores para enviarlos en el formato adecuado.
- Se transmiten los datos por el puerto serie en el orden establecido y con el contenido adecuado. Se verifica que los mensajes se transmiten en el mismo orden que se capturan.
- El sistema espera a que el módulo UART esté libre para enviar un nuevo dato por el puerto serie.
- La máquina de estados implementada funciona adecuadamente.
- El contador de mensajes aumenta su valor tras el envío de cada mensaje
- La salida LED_EMPTY está a nivel alto cuando las memorias no contienen datos
- La salida LED_MSG_LOST se activa y permanece activa cuando se recibe un nuevo mensaje y las memorias están llenas. Permite identificar que la perdida de mensajes capturados será reconocible por el sistema.

- Memorias FIFO

Para este módulo, se ha creado un banco de pruebas que permite conocer a fondo el comportamiento del módulo y validar su funcionamiento. Se comprueba que:

- Almacena los mensajes recibidos por su entrada DIN cuando se activa la entrada WR_EN.
- Los datos almacenados se extraen mediante la salida DOUT en el orden que han sido almacenados. Siendo estos visibles cuando la salida VALID está a nivel alto
- Un dato almacenado se extrae de la memoria cada vez que se pone a nivel alto la entrada RD_EN.
- La salida EMPTY está a nivel alto mientras no hay datos almacenados.
- La salida FULL se pone a nivel alto cuando se alcanza la capacidad máxima de la memoria.

- Conversor de Hexadecimal a ASCII

En este caso, el banco de pruebas diseñado contiene un proceso que verifica que la conversión de cada posible valor de entrada se realiza de forma adecuada.

Éste proceso envía al conversor cada uno de los posibles valores a recibir y verifica que la salida DOUT_VALID está a nivel alto cuando finaliza la conversión y que el valor de ésta es el adecuado.

- Conversor de binario a BCD

Se ha implementado un banco de pruebas que verifica que el componente:

- Almacena adecuadamente el valor recibido por la entrada DIN cuando se activa la entrada ENABLE.
- La máquina de estados implementada funciona adecuadamente y se producen las transiciones entre estados esperadas.
- El contador de bits desplazados funciona adecuadamente.
- La salida DOUT muestra el valor, expresado en BCD, del dato recibido por su entrada.
- La salida DOUT_VALID se pone a nivel alto cuando la conversión ha finalizado.

- UART

Para comprobar que este módulo y sus componentes funcionan adecuadamente y, dado que no han sido específicamente diseñados para este proyecto, conocer su funcionamiento en profundidad, se ha implementado un banco de pruebas de simulación que verifica los siguientes aspectos:

- La salida de transmisión serie TXD_PAD_O permanece a nivel alto cuando el sistema está en reposo.
- Cuando la entrada LOADTX se pone a nivel alto, el sistema comienza una transmisión.
- La transferencia comienza con un bit de inicio fijado a nivel bajo.
- Los datos son transferidos comenzando por el LSB del byte recibido en la entrada BYTE2PC y terminando por el MSB de éste. Se verifica que la velocidad de transmisión coincide con la deseada.
- La transmisión de datos finaliza con el bit de parada a nivel alto.
- El módulo es capaz de recibir datos a través de su entrada RXD_PAD_I muestreando la línea según marque el contador de recepción.
- Se verifica que el byte recibido coincide con el enviado y que la salida RXAV se pone a nivel alto cuando la recepción ha finalizado.
- El módulo indica que está ocupado y no puede recibir más datos mediante la activación de su salida TXBUSY.

- I2C Sampler

Otro banco de pruebas ha sido diseñado para comprobar que los datos se capturan adecuadamente y se identifican los posibles fallos expuestos anteriormente. Dicho banco incluye los simuladores de maestro y esclavo I2C para simular las transferencias. Se han verificado los siguientes aspectos:

- Se identifica el comienzo de una transmisión reconociendo la condición de inicio (bajada de SDA mientras SCL está a nivel alto).
- Se captura cada bit transferido durante el tiempo a nivel alto del reloj SCLK. Se verifica que los bits se almacenan adecuadamente en el registro de desplazamiento.
- El sistema es capaz de capturar la dirección del esclavo transferida y el tipo de operación a realizar. Al mismo tiempo se verifica que los 7 bits de dirección y el bit de operación se almacenan en el registro de 8 bits de manera adecuada. Cuando esta captura finaliza el sistema activa la salida NEW_BYTE para que la información pueda ser almacenada.
- El sistema captura la señal de asentimiento de direccionamiento enviada por el esclavo. Cuando comienza la recepción del primer bit de datos el sistema activa su señal de NW_EOM y envía el valor de la señal de asentimiento a su salida ACK.
- Cada uno de los bytes transmitidos por el bus I2C se captura adecuadamente y se envía a la salida BYTE_RECEIVED, fijándose a nivel alto la salida NEW_BYTE para indicar el final de la captura.
- El sistema es capaz de identificar si un byte es el último transferido en un mensaje e indicarlo mediante su salida EOM. La salida está a nivel alto en caso positivo y a nivel bajo en caso negativo.
- Si tras el envío de la dirección el esclavo no envía la señal de asentimiento, la salida ACK toma un nivel alto indicando que no se ha direccionado correctamente

- Si la transmisión se corta con una condición de parada durante el envío de un byte, la salida DOUT refleja los bits recibidos hasta el momento en las posiciones más altas y ceros en el resto de posiciones correspondientes a los bits no recibidos. Las salidas ACK y EOM se fijan a nivel alto indicando que la transmisión se ha cortado sin finalizar.
- Si la transmisión se corta sin condición de parada el sistema mantiene su salida RUNNING activa para indicarlo.

- Bloque de control I2C

Este bloque ha sido probado de manera similar al bloque de control del analizador SPI por lo que muchas comprobaciones son semejantes. Se han verificado los siguientes aspectos:

- La memoria de 8 bits almacena el byte capturado cuando la entrada NEW_BYTE tiene un nivel lógico alto.
- La memoria de 2 bits almacena la señal de asentimiento y la información de final de mensaje en el momento adecuado.
- Si las memorias tienen datos almacenados, la máquina de estados comienza la transferencia de un mensaje.
- Los conversores reciben los datos adecuados en cada momento y las conversiones se realizan como deberían.
- La salida serie transfiere los datos en el formato adecuado, comenzando con la cabecera del mensaje.
- El contador de mensaje aumenta su valor al finalizar cada envío.
- La salida LED_EMPTY refleja que las memorias están vacías cuando no quedan mensajes almacenados.
- Si las memorias están llenas y se recibe un nuevo byte, la salida LED_MSG_LOST pasa a nivel alto. Si se recibe otro mensaje y éste sí es almacenado, dicha salida permanecerá activa indicando que se ha perdido algún mensaje.
- Se solicita una nueva transferencia al módulo UART únicamente cuando éste está disponible para ello. Se comprueba que mientras su salida BUSY esté activa no se solicitan nuevas transferencias.

Pruebas del analizador completo:

- TOP_SPI

Se ha implementado un banco de pruebas de simulación capaz de comprobar que:

- El bloque SPI_sampler captura los mensajes enviados por un proceso que simula transmisiones entre un bloque maestro y un esclavo.
- El bloque de control SPI almacena los mensajes capturados adecuadamente en las memorias y los envía a través de la salida serie con el formato adecuado.
- Las conexiones entre ambos bloques se comportan como se espera.

- TOP_I2C

Se ha implementado un banco de pruebas de simulación que verifica los siguientes aspectos:

- El bloque I2C_sampler captura los datos intercambiados entre dispositivos maestros y esclavos que han sido diseñados como modelos de simulación I2C.

- El bloque de control I2C recibe los mensajes capturados y los almacena adecuadamente en las memorias. Cuando algún mensaje ha sido almacenado el sistema comienza la transmisión a través del puerto del puerto serie y se verifica que los datos se envían correctamente.
- La interfaz de ambos módulos se ha conectado correctamente.

A modo de ejemplo de las pruebas realizadas, la siguiente imagen refleja el interfaz del simulador ISim mientras se comprueba el funcionamiento del bloque I2C_Sampler. En ella puede observarse como la salida BYTE_RECEIVED (marcada en rojo) toma en primer lugar el valor de la dirección de esclavo que envía el maestro (marcada en azul) y el tipo de operación, por lo que el último bit toma el valor de '0'. A continuación, la salida BYTE_RECEIVED captura el byte enviado por el maestro (marcado en amarillo). Posteriormente, se direcciona al mismo esclavo, pero la operación es de escritura por lo que la salida BYTE_RECEIVED toma el valor de la dirección del esclavo pero el LSB tiene el valor '1' indicando operación de lectura. Finalmente, la salida toma el valor del byte enviado por el esclavo (marcado en violeta) capturando el byte que ha recibido:

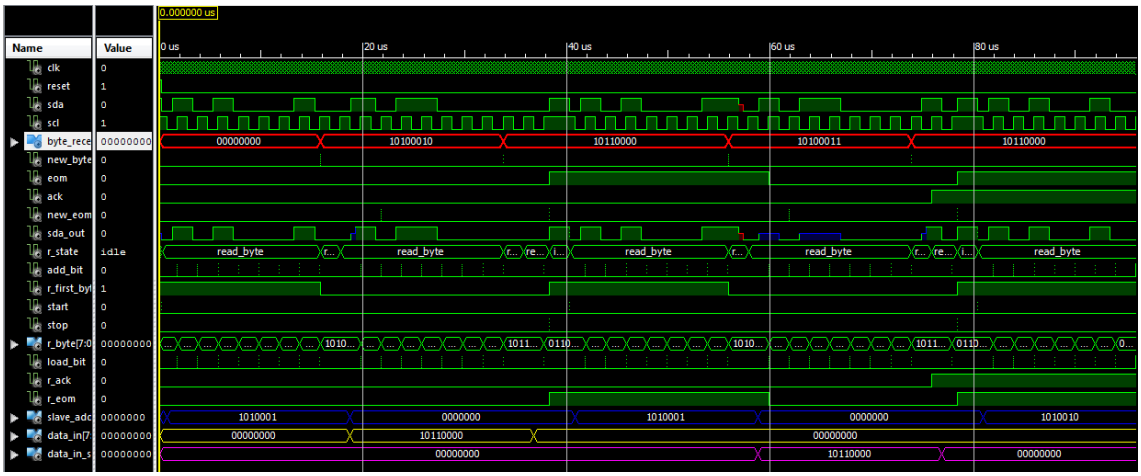


Figura 50: Captura de simulación del bloque I2C_Sampler

4.3 Pruebas a nivel Hardware

Finalizada la verificación mediante simulación de los componentes es necesario realizar diversas pruebas para comprobar que el sistema diseñado funciona en un equipo real. Para ello, ha sido necesario emplear diversos elementos que nos permitan componer un entorno real de trabajo del sistema.

A continuación se detallarán el entorno de desarrollo empleado y los elementos utilizados para realizar las pruebas a nivel físico del sistema y, posteriormente, se describirán dichas pruebas realizadas.

4.3.1 Entorno de desarrollo

Dado que, como se ha comentado previamente en la definición de las características del diseño (ver sección: 2.1.1), el diseño ha sido realizado empleando el lenguaje de descripción de hardware VHDL, ha sido necesario emplear un entorno de desarrollo capaz de soportar la implementación con dicho lenguaje.

Existen diversos entornos, desarrollados por diferentes fabricantes, que permiten sintetizar un diseño que ha sido descrito empleando el lenguaje VHDL. La mayoría de dichos entornos son desarrollados por los propios fabricantes de FPGAs de manera que el entorno empleado se decide tras seleccionar el dispositivo sobre el que se va a implementar.

Sin embargo, dado que el lenguaje VHDL es un estándar, existe la posibilidad de portar los diseños a diferentes entornos, donde el propio sistema se encargará de hacer la "traducción" adecuada del código para obtener una implementación que se ajuste al dispositivo donde se va a integrar el diseño.

Los principales fabricantes de FPGAs son Xilinx y Altera que disponen de diferentes dispositivos categorizados por familias que presentan diversas arquitecturas empleadas para diferentes aplicaciones. También, existen otras empresas con menor presencia en el mercado como son Actel o la europea Atmel que poseen algunos dispositivos muy eficientes para determinadas aplicaciones.

En el caso expuesto en este proyecto, se ha optado por emplear un dispositivo desarrollado por Xilinx como sistema de partida para poder estimar los recursos consumidos por el diseño y verificar el correcto funcionamiento de éste a nivel hardware.

Concretamente, el diseño se implementará en una FPGA Spartan 3-E de Xilinx, la cual será presentada más adelante en el apartado de verificación hardware (ver sección 4.3.2.1). Por ello, el entorno empleado para el desarrollo será el facilitado por el fabricante Xilinx, empleándose la versión descrita en el siguiente apartado.

4.3.1.1. Xilinx ISE 13.4

El fabricante Xilinx ofrece en página web, consultable en el punto de la bibliografía Web del fabricante Xilinx, diferentes versiones de su entorno de desarrollo ISE.

En este caso, se ha decidido emplear la versión 13.4 de dicho entorno ya que es una versión robusta presentada hace tiempo por lo que está muy depurada y no presenta errores. Además, se trata del entorno de desarrollo empleado en diversos trabajos anteriores por lo que se está familiarizado con él.

A continuación, se describirán las principales características del entorno de desarrollo Xilinx ISE 13.4 y los pasos necesarios para obtener el fichero binario de programación (archivo .bit) que permite integrar el diseño en la FPGA.

4.3.1.2. Características Xilinx ISE 13.4

Este entorno de desarrollo incluye diversas herramientas que permiten configurar diferentes opciones:

- ChipScope Pro: Analizador de circuitos que permite comprobar el comportamiento de estos una vez implementado el diseño en la FPGA
- Xilinx Platform Studio (XPS): herramienta empleada para la conexión y configuración de procesadores embebidos.
- Xilinx Software Development Kit (SDK): Entorno de desarrollo de software para el procesador embebido.

- Xilinx PlanAhead: herramienta empleada para la visualización de la implementación RTL y el análisis de tiempos que permite configurar restricciones del diseño y las zonas de la FPGA en las que se va a implementar cada parte del diseño.
- Project Navigator: Herramienta principal del entorno que permite sintetizar el proyecto y obtener el archivo de programación generado a partir de este.

Además como principales características este entorno incluye:

- Permite la síntesis y el análisis de los diseños realizados empleando los lenguajes VHDL y Verilog, empleados para la descripción de hardware.
- Ofrece la posibilidad de realizar análisis temporales del diseño.
- Presenta un analizador de diagramas RTL que facilita el análisis de la síntesis realizada.
- Incluye un simulador que facilita la depuración del diseño realizado pudiendo someter a éste a diferentes estímulos.
- Tecnología de sincronización de reloj inteligente. Esta función permite reducir el consumo dinámico de potencia en un 30% respecto a otras versiones.
- Flujo de diseño intuitivo gracias a la interfaz de usuario. El seguimiento del flujo de diseño se hace de forma sencilla.
- Incluye diferentes asistentes de configuración para el manejo de módulos de propiedad intelectual, analizadores, SoC, etc. Esto permite acelerar el desarrollo del diseño no siendo necesario describir todos los módulos empleados.
- Capacidad de realizar diseños jerárquicos. De este modo el diseño puede descomponerse en bloques facilitando el desarrollo.

4.3.1.3. Proceso de generación del archivo binario de programación

Una vez diseñados los diferentes módulos que componen el diseño es necesario seguir una serie de pasos para obtener el fichero binario de programación que permitirá implementar el diseño en la FPGA seleccionada:

- Definir el archivo de restricciones del diseño (archivo .ucf). Para ello, puede emplearse la herramienta Xilinx PlanAhead para seleccionar las restricciones temporales del diseño y definir los pines de la FPGA a los que se conectarán cada una de las entradas y salidas del diseño. El archivo generado para este diseño puede observarse en el anexo 11.2.
- Sintetizar el proyecto para obtener el equivalente RTL del diseño. Para ello debe usarse el sintetizador XST presente en el navegador de proyecto en la ventana de procesos.
- Implementar el diseño. Desde el navegador de proyectos puede realizarse la implementación del diseño que se compone de 3 partes:
    a. Traducción: el entorno de desarrollo “traduce” los archivos resultantes de la síntesis (archivos .ngc) para comprobar los elementos de la FPGA que requieren ser utilizados.
    b. Mapeo: el sistema define mediantes diversos algoritmos las regiones de la FPGA donde se va a implementar cada parte del diseño.

c. Posicionamiento y rutado: Se decide la posición que ocupará cada elemento y las conexiones que será necesario realizar entre ellos.

- Finalizados estos pasos, se genera el archivo binario de programación que se empleará para configurar la FPGA de modo que se alcance la funcionalidad esperada.

La imagen mostrada a continuación muestra la interfaz gráfica del navegador de proyectos desde donde se gestionan todas estas acciones:

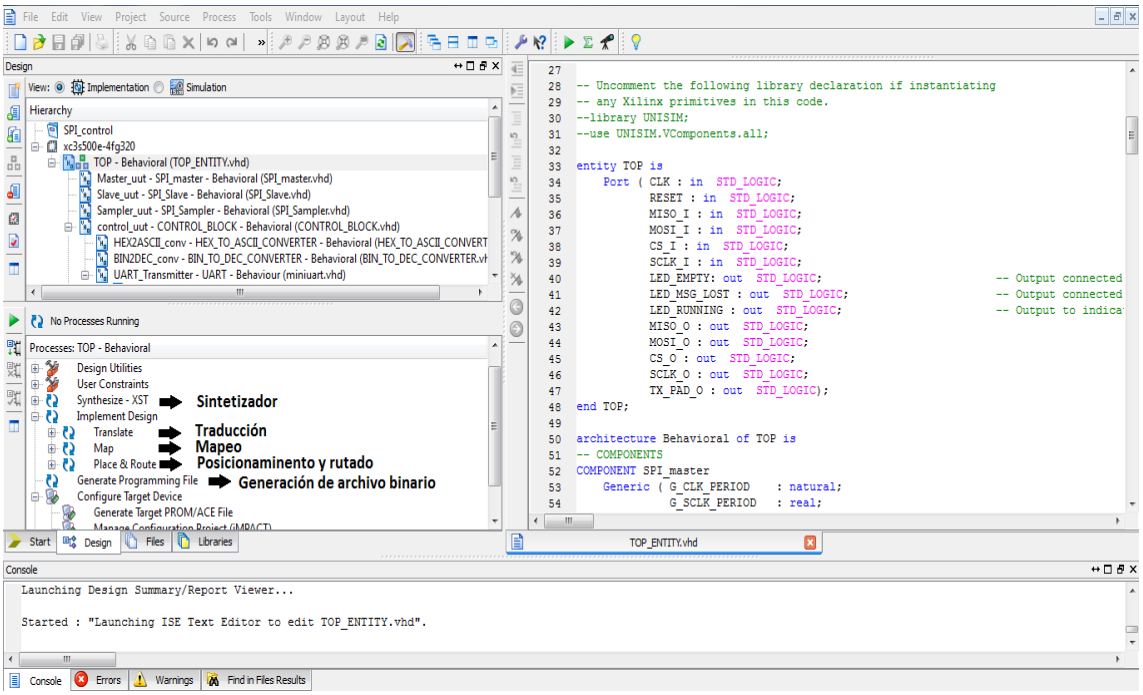


Figura 51: Interfaz del navegador de proyectos ISE 13.4

4.3.2 Elementos empleados en la verificación

Dado que el diseño desarrollado será empleado en una FPGA, se ha seleccionado un modelo común para realizar las pruebas. Además, considerando que la transferencia de los datos hacia el ordenador se realiza mediante un puerto serie poco común en los ordenadores actuales, ha sido necesario emplear un conversor de RS-232 a USB para poder recibir los datos a través de un puerto del ordenador. Por último, para poder visualizar los datos recibidos, se debe disponer de un software capaz de leer los datos del puerto serie y mostrarlos en la pantalla del ordenador. Todos estos elementos serán descritos en los apartados posteriores.

4.3.2.1. Xilinx SPARTAN 3-E

La FPGA empleada para realizar las pruebas ha sido facilitada por el tutor del proyecto. Se trata de una FPGA Xilinx perteneciente a la familia Spartan 3-E montada sobre una placa de evaluación que posee diversos periféricos.

Concretamente, la FPGA empleada es el dispositivo XC3S500E, empleándose el paquete FG320 que incluye 320 pines.

Este dispositivo está integrado en una PCB que incluye diversos periféricos y recibe el nombre de Spartan 3E Starter Kit Board. La guía de usuario de esta placa puede consultarse en la referencia [REF-4] Guía de usuario de la FPGA Spartan 3E Starter kit Board. Además, el anexo

11.1 incluye la lista de características de la FPGA Spartan 3-E donde pueden observarse todos los periféricos incluidos.

A continuación se detallarán los periféricos de esta placa empleados en el diseño:

Periféricos empleados

- Fuente de reloj

Como fuente de reloj empleada para sincronización del diseño se ha empleado el oscilador de 50 MHz montado en la placa. Este oscilador ofrece una señal de reloj cuyo ciclo de trabajo puede variarse entre el 40 y el 60%. Ofrece una precisión de ±2500 Hz o lo que es lo mismo, ±50 ppm (partes por millón) lo cual, considerando la elevada frecuencia de salida, es una precisión alta que evitará que se produzcan problemas en la sincronización del diseño.

- Conectores de expansión

Para conseguir que las pruebas realizadas se asemejen lo máximo posible a un entorno real, se emplean los conectores de expansión disponibles en la placa como pines de salida de las líneas de transferencia de los simuladores. En el caso de los simuladores SPI, las salidas corresponderán a las líneas MOSI, MISO, SCLK y CS que componen la interfaz del bus SPI. Para los simuladores I2C, se extraen las líneas SDA y SCL que forman la interfaz del protocolo I2C.

Además, estos conectores se emplean como puertos de entrada de los analizadores de modo que el analizar se conecta a las salidas adecuadas de los simuladores asemejándose a la conexión que existiría en un sistema real.

Estos conectores se presentan como un conector hembra que incluye 6 pines, de los cuales 4 pueden emplearse como entradas/salidas y los otros dos corresponden a la tensión de alimentación de 3,3 V que ofrece la placa y la masa del conector.

- Puerto serie RS-232

Para realizar las transferencias desde la FPGA al ordenador se emplea el puerto serie que ésta incluye. Concretamente, la placa empleada dispone de un conector RS-232 macho (denominado DTE por sus siglas en inglés *Data Terminal Equipment*) y otro conector hembra (denominado DCE por sus siglas en inglés *Data Comunications Equipment*).

En nuestro caso se ha empleado el terminal DCE porque el conversor RS-232 a USB empleado presenta un conector macho en su conector RS-232.

Este terminal posee 9 pines, de los cuales, en el caso de esta placa, uno es empleado para la transmisión otro para la recepción de datos. Un tercer pin se emplea como conexión a masa del conector y los demás no son utilizados.

En nuestro caso, únicamente se configurará el pin empleado para la transmisión de datos ya que el diseño realizado no emplea el receptor serie.

- Leds

Como método de identificación de fallos y comprobación del funcionamiento, como ya se ha comentado anteriormente, el diseño emplea una serie de salidas que se conectarán a unos LEDs.

La placa utilizada dispone de un conjunto de 8 LEDs de montaje superficial que poseen un terminal conectado a masa y el otro conectado a un pin de la FPGA mediante una resistencia de 390Ω encargada de limitar la corriente. Esto permite que el LED se encienda poniendo un nivel alto de tensión en el pin de la FPGA.

El diseño realizado empleará 6 de estos LEDs, 3 para cada analizador como se ha explicado previamente.

- Interruptores (*switches*)

Por último, para controlar el reinicio del sistema y la selección del analizador a utilizar, se han empleado los interruptores de deslizamiento disponibles en la placa empleada.

Dichos interruptores conectan el pin de la FPGA al que se unen a 3,3 V cuando se encuentran en su posición de encendido haciendo que la FPGA identifique un nivel lógico alto. Cuando el interruptor está en su posición de apagado, el interruptor conecta el pin de la FPGA a tierra de modo que ésta identifica un nivel lógico bajo en su entrada.

La imagen presentada a continuación y que ha sido extraída de la guía de usuario de la placa ([REF-4] Guía de usuario de la FPGA Spartan 3E Starter kit Board) identifica mediante recuadros los periféricos de ésta empleados:

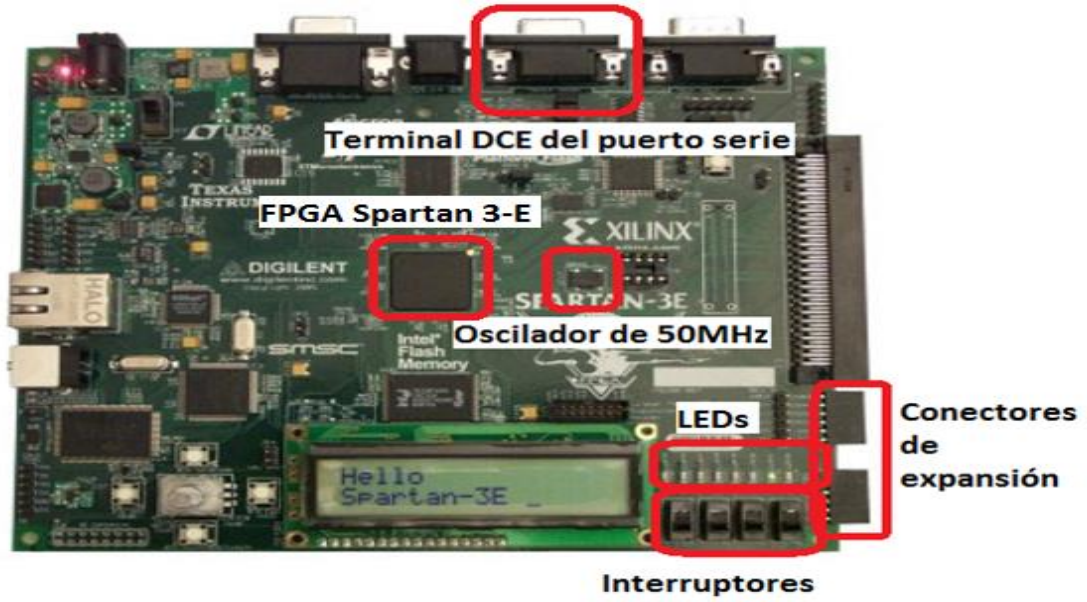


Figura 52: Disposición de periféricos en la FPGA Spartan 3-E

4.3.2.2. Cable adaptador RS-232 a USB

Dado que los ordenadores actuales no disponen de un puerto serie RS-232, se ha empleado un adaptador de USB a RS-232 que permite conectar el terminal DCE de la placa empleada a uno de los puertos USB disponibles en el ordenador. El puerto USB empleado será identificado en el ordenador como un puerto serie, es decir, con la nomenclatura COM.

El adaptador empleado es un modelo ICUSB232 fabricado por StarTech.com y sus características pueden consultarse en el anexo 11.3 Ficha técnica del adaptador de USB a RS-232 ICUSB232.

4.3.2.3. Software de comunicación

Para poder analizar los datos recibidos en el ordenador es necesario disponer de un software capaz de leer el puerto del ordenador al que se conecta el diseño y mostrar en pantalla los datos recibidos.

Existen diversos programas, conocidos como emuladores de terminal, capaces de realizar esta función como son Tera Term, X-CTU o PuTTY. Estos softwares permiten configurar la velocidad a la que se efectúa la transmisión y los parámetros como longitud del mensaje, número de bits de parada o paridad que definen la transferencia.

En este caso, se ha empleado el software PuTTY ya que se trata de un programa de código libre y que presenta una versión portable por lo que no requiere instalación previa en el ordenador.

La siguiente imagen recoge la configuración empleada para recibir los datos desde el diseño realizado:

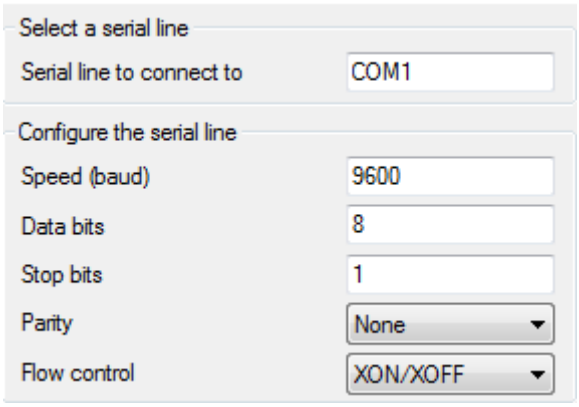


Figura 53: Configuración de la transmisión serie en PuTTY

Una vez definidos los elementos empleados, a continuación de detallaran el conjunto de pruebas realizadas al sistema estudiado.

4.3.3 Pruebas realizadas

Tras implementar el diseño junto a los modelos de simulación en la FPGA usando las herramientas descritas y realizar las conexiones necesarias, se han realizado una serie de pruebas que permiten corroborar el funcionamiento del diseño ante varias situaciones, tanto en el caso del analizador SPI como en el del analizador I2C. A continuación se describirán las pruebas realizadas a cada uno de los analizadores:

- Analizador SPI:
  - Se verifica que el diseño captura tramas enviadas a través del protocolo SPI empleando diversas velocidades de transmisión.
  - Se comprueba que el sistema captura mensajes con diferentes longitudes. En el ordenador se visualizan únicamente los datos recibidos como se ha explicado anteriormente.
  - Mediante la desconexión durante un envío de la entrada CS, se verifica que el sistema identifica el corte de la transmisión manteniendo encendido el LED de RUNNING.

  - Se verifica que los datos del mensaje mostrados en el ordenador (longitud en bits, polaridad y fase del reloj) coinciden con el modo de transmisión empleado en cada caso.
  - Empleando una velocidad de transmisión muy alta (12,5 Mbaudios) en el bus SPI y enviando un gran número de mensajes (en torno a los 1500) se comprueba que el diseño satura las memorias. Se verifica que el LED MSG_LOST permanece activo aunque se almacenen mensajes posteriores al perdido.
  - Se verifica que, una vez transferidos al ordenador todos los mensajes capturados, el sistema activa su LED de EMPTY para indicar que las transferencias han finalizado.
- Analizador I2C:
  - Se comprueba que sistema es capaz de capturar mensajes transferidos empleando la velocidad definida en cada uno de los modos del protocolo I2C, desde los 100 Kbits/s hasta los 5Mbits/s.
  - Se verifica que diseño es capaz de capturar mensajes de distinta longitud (en bytes) y que los mensajes se muestran adecuadamente en la pantalla, conteniendo todos los datos transferidos.
  - Mediante una configuración especial de los simuladores de maestro y esclavo I2C donde la transmisión se corta con un stop durante el envío de un byte, se verifica que el sistema identifica el fallo. Se confirma que el byte cortado se refleja en el ordenador conteniendo los bits que han sido capturados y ceros en las posiciones que no se han llegado a transferir. Se verifica también que dicho byte se muestra como final d mensaje y refleja que el receptor no ha enviado señal de asentimiento.
  - Empleando otra configuración especial de los simuladores en la que la transmisión se interrumpe durante un envío sin condición de parada, se verifica que el diseño identifica este fallo manteniendo encendido su LED de RUNNING para indicar que la transmisión no ha finalizado correctamente.
  - Usando de nuevo una velocidad de transmisión muy elevada (de 5Mbits/s) y enviando un gran número de mensajes a través del protocolo I2C, se comprueba que el diseño satura las memorias y enciende el LED de MSG_LOST para indicar que algún mensaje se ha perdido.
  - Se verifica que, una vez que todos los mensajes capturados han sido transferidos al ordenador, el sistema enciende su LED de EMPTY para indicar que no hay más mensajes capturados por enviar. Se comprueba que el diseño pasa al estado de reposo a la espera de la captura de nuevos datos a través del protocolo.

4.4 Resultados

Finalizadas todas las pruebas, tanto a nivel de simulación como a nivel físico, se pueden analizar los datos obtenidos de estas.

Considerando que todas las pruebas han sido superadas con éxito por el diseño, se puede afirmar que éste será capaz de conectarse a un bus físico y enviar al ordenador los datos que han sido transmitidos a través de éste.

El diseño es capaz de analizar transmisiones independientemente de la velocidad a la que estas se realicen. Además, es posible capturar mensajes de diferentes longitudes convirtiendo al sistema en un diseño robusto.

La implementación realizada en el diseño permite, además de capturar las tramas, identificar los fallos en la transmisión expuestos anteriormente. Esto aporta fiabilidad al analizador dotándole de la capacidad de usarse para depurar dispositivos maestros y esclavos que se conecten a ambos buses.

Considerando la capacidad de las memorias empleadas y las pruebas a las que han sido sometidas, se puede afirmar que el dispositivo es capaz de capturar grandes cantidades de información, siendo difícil que éste se sature.

Como ejemplo, las figuras expuestas a continuación reflejan los mensajes capturados mediante el analizador SPI y el analizador I2C respectivamente durante algunas de las pruebas realizadas:

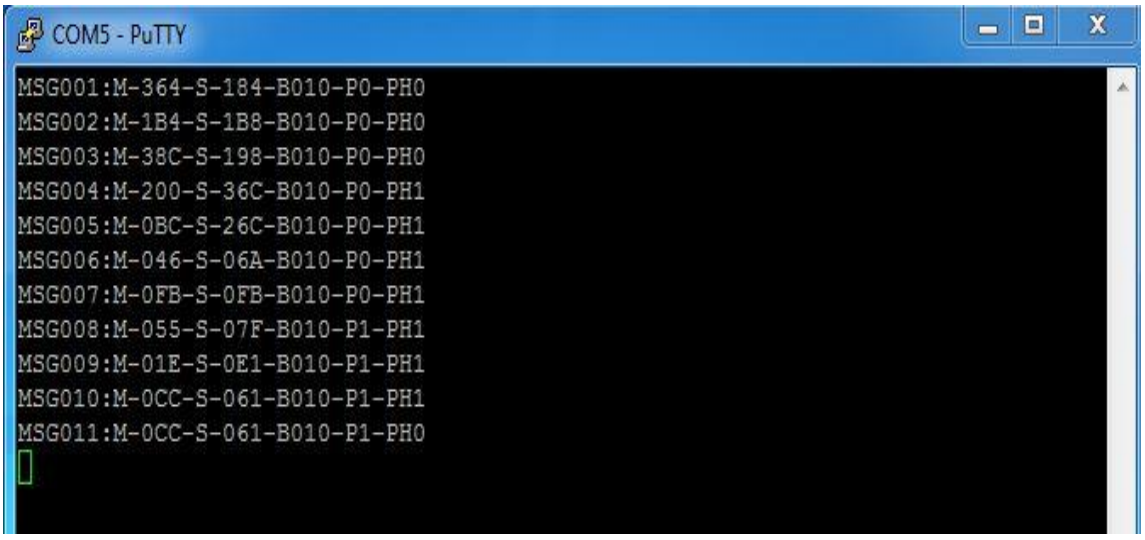


Figura 54: Captura de mensajes del analizador SPI

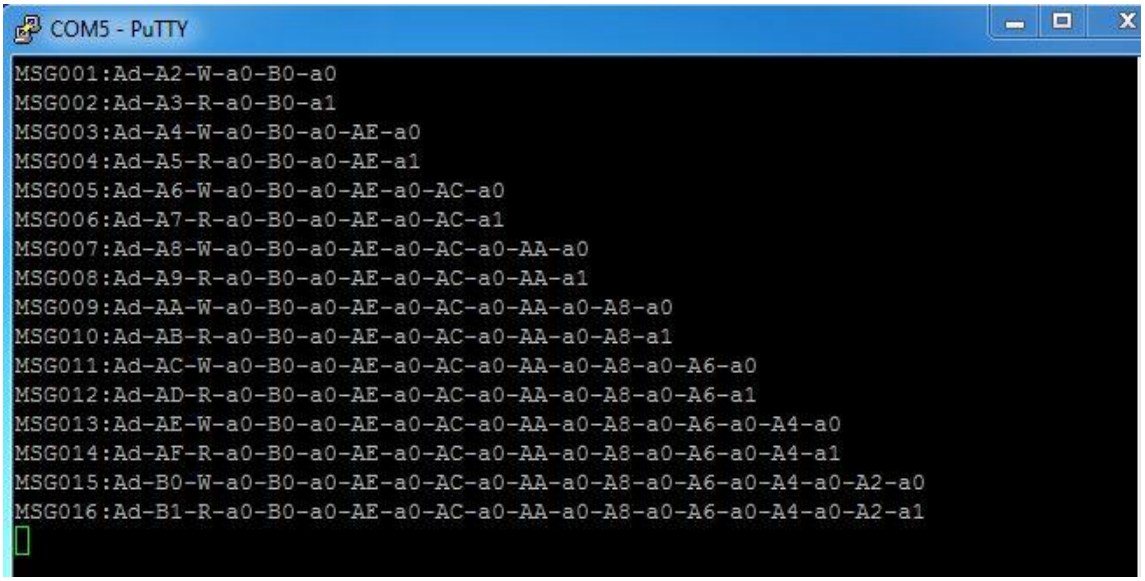


Figura 55: Captura de mensajes del analizador I2C

5. Informes de resultados de síntesis

Una vez finalizado el desarrollo del proyecto, éste es sintetizado como se indica en la sección 4.3.1.3. El sintetizador ofrece una serie de resultados de la síntesis dónde pueden verse diferentes datos referentes a la ocupación de recursos lógicos del dispositivo en el que se implementará el diseño, las características temporales que éste presentará o información referente al proceso de rutado del diseño.

Esta sección define los resultados de ocupación y el análisis temporal del diseño, especificando los resultados de cada analizador y del conjunto de ambos, y presenta algunas conclusiones a cerca de ellos.

5.1 Resultados del análisis de ocupación

Respecto a la ocupación de recursos del dispositivo donde se va a implementar el diseño, usando como referencia la citada FPGA de Xilinx Spartan-3E XC3S500E FG320, el entorno de Xilinx ISE 13.4 ofrece los resultados de ocupación expuestos en la imagen siguiente:

	LUTs	**Flip Flops**	**Slices**	**Bloques RAM**
Analizador SPI	1429	940	1024	16
Analizador I2C	383	301	301	2
Conjunto de analizadores	1796	1239	1316	18
Disponibles en Spartan 3-E	9312	9312	4656	20

Tabla 37: Resultados de ocupación de lógica del diseño

En la imagen expuesta se puede observar que la ocupación de los grupos de biestables disponibles en el dispositivo es de 940 para el analizador SPI, 301 para el analizador I2C, y 1239 en el caso del sistema que los une. Este dato nos indica que el diseño no requiere el empleo de un elevado número de biestables para implementar la función buscada. La mayor parte de estos biestables se generan a partir de los diferentes registros empleados en el diseño, lo cual explica que el analizador SPI requiera un mayor número ya que los registro son más grandes por su capacidad de capturar 128 bits en cada mensaje.

A continuación, puede observarse que se emplearán 1429 LUTs (*Look Up Table*) para el analizador SPI y 383 para el analizador I2C, empleando un total de 1796 cuando se sintetiza el conjunto de ambos. Este resultado nos indica que la lógica desarrollada en el diseño no requiere un elevado empleo de funciones lógicas y que las empleadas no generan un gran número de combinaciones ya que el diseño realizado busca la optimización de los recursos. Estas LUTs contienen un conjunto de puertas lógicas interconectadas entre ellas y su uso permite obtener el resultado de una operación lógica de una manera rápida.

Posteriormente, se observa que el diseño requiere 1024 *Slices* para el analizador SPI, 301 para el analizador I2C y un total de 1316 para el conjunto de ambos. Estos *Slices* se refieren a las unidades de la FPGA que contienen un número de biestables, LUTs y multiplexores que puede variar en función de la familia de FPGAs utilizada. Observando los resultados anteriores de

biestables y LUTS empleados, se observa que el software encargado de mapear el diseño en el dispositivo no ha realizado un gran esfuerzo ya que el porcentaje de utilización no es muy elevado. Sin embargo, tampoco ha sido una operación óptima ya que ha tenido que ha empleado un total de *Slices* superior al porcentaje de biestables y LUTs requeridos, lo cual indica que no todos los recursos de cada *Slice* ocupado son empleados.

Cabe destacar que esto no se debe a una mala práctica de diseño, si no que el software de mapeo decide implementar cada parte de código en diferentes regiones de la FPGA para que queden separadas aunque no se empleen todos los recursos de cada zona.

Además, puede observarse que el total de *Slices* empleadas contiene lógica relacionada, hecho que indica que la afirmación anterior es correcta.

Respecto al número total de LUTs de 4 entradas, el informe de síntesis también indica que se empleará la mayor parte de éstas para lógica como se ha comentado anteriormente. La parte restante ha sido utilizada como accesos a puntos internos de los *Slice* en los que se encuentra la LUT. Este número es muy bajo y su presencia se debe a casos en los que no es posible acceder a un componente interno del *Slice* de manera directa.

El número de buffers de entrada/salida empleados es de 16 en el caso del conjunto de analizadores, usándose uno para cada uno de los puertos que incluye el diseño. La FPGA dispone de 232 pines de sus 320 que pueden ser usados como entradas o salidas por lo que el diseño ocupa únicamente el 6% de éstos.

Los bloques de memorias RAM empleados suponen el mayor porcentaje de ocupación del diseño ya que éste requiere 18 de los disponibles para sintetizar el conjunto. La mayor parte de esta ocupación se debe a la presencia de las memorias FIFO del diseño, especialmente las del analizador SPI que son las más grandes, ya que se implementan como bloques de RAM. En caso de que el diseño se incluya en otro proyecto, sería recomendable reducir la profundidad de dichas memorias para permitir liberar bloques de RAM en el dispositivo.

5.2 Resultados del análisis temporal

En lo que se refiere a las características temporales del diseño, el informe muestra diferentes resultados. Los más importantes son:

- Resultado del análisis del periodo del reloj CLK:

	Tiempo entre biestables y frecuencia máxima de reloj	**Tiempo mínimo entre pin de entrada y biestable**	**Tiempo máximo entre biestable y pin de salida**	**Tiempo de pista combinacionales**
Analizador SPI	11.535ns => 86,694 MHz	6.808ns	5.693ns	No existe
Analizador I2C	7.610ns => 131.406MHz	4.714ns	6.317ns	No existe
Conjunto de analizadores	11.535ns => 86,694 MHz	6.808ns	6.326ns	6.497ns

Tabla 38: Resultados temporales del diseño

Este informe indica que el sistema sería capaz de soportar un reloj de hasta 86,694 MHz, valor muy superior a los 50 MHz empleados por lo que el diseño no tendrá problemas de sincronización. En caso de implementar únicamente el analizador I2C, podría emplearse un reloj superior ya que la distancia entre biestables relacionados del sistema es menor que en el caso del analizador SPI, sistema que actúa como limitante en el conjunto del diseño.

Además, el tiempo máximo entre un cambio en la entrada y su reflejo en el biestable que la registra es de 6.808ns por lo que no supera el periodo de un ciclo de reloj (20ns) y no debe suponer problemas en el diseño.

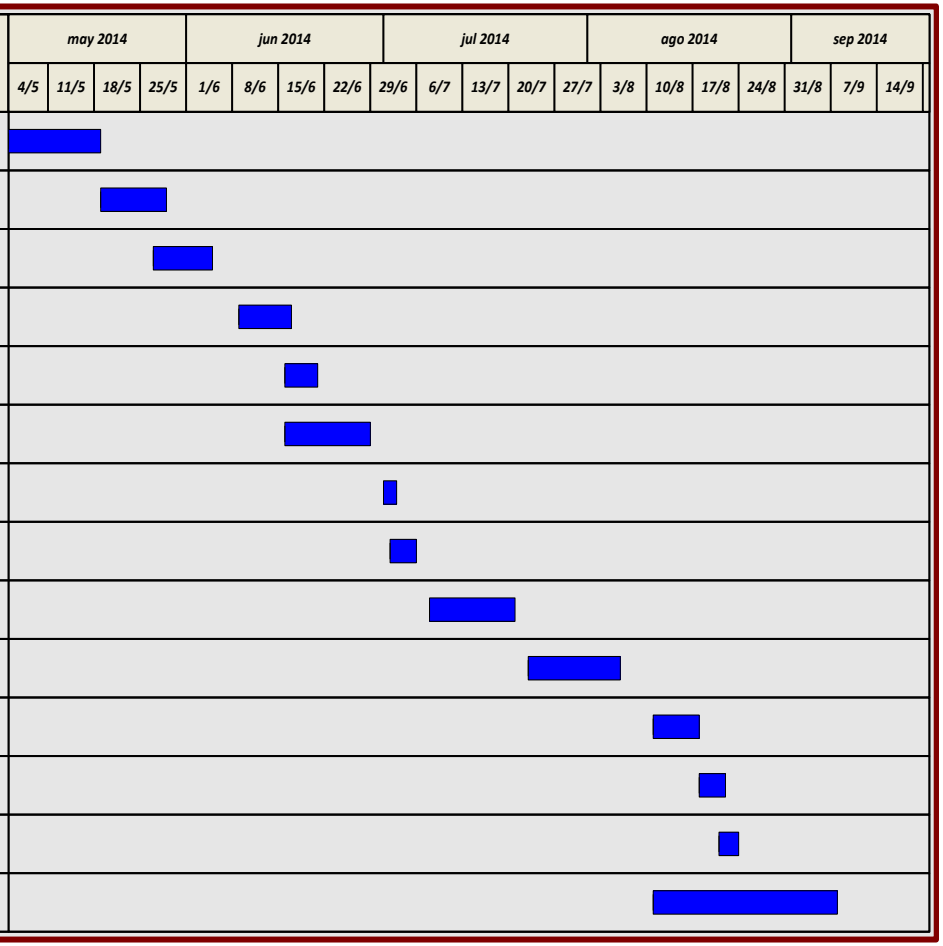
El retraso máximo entre la salida de un biestable y un pin de salida se sitúa en los 6.326ns en el caso del conjunto de analizadores, hecho que tampoco supondrá problemas en el comportamiento del diseño.

Por último, el retraso máximo entre pistas combinacionales es de 6.497ns en el caso del conjunto de analizadores, valor que no supondrá problemas en el diseño. Además, observando el informe detallado, se identifica que este retraso se produce únicamente en el caso de implementar ambos analizadores juntos. Este retraso se produce entre la entrada SELECT_PROTOCOL y la salida RS232_TX_PAD por lo que el retraso es producido por el multiplexor que se emplea para controlar el envío por el puerto serie de los datos del analizador SPI o el I2C. Considerando que la selección de analizador a utilizar se realizará al iniciar el funcionamiento, este retraso no debería suponer problemas en el funcionamiento.

Una vez descrito todo el diseño, realizadas las pruebas de éste y comprobados los resultados obtenidos, se describirán mediante un diagrama de Gantt los tiempos empleados en cada una de las fases de desarrollo del proyecto. A continuación, se presentará el entorno socio-económico en el que el diseño puede ser empleado así como un presupuesto que detalla los costes requeridos para la realización de dicho proyecto. Posteriormente, se presentarán una serie de conclusiones obtenidas tras el diseño y se mencionarán una serie de posibles líneas de trabajo futuro sobre el diseño que ha sido realizado en este proyecto.

6. Fases de desarrollo del proyecto

Id.	Nombre de tarea	Comienzo	Fin	Duración
1	Estudio y análisis de protocolos SPI e I2C	05/05/2014	18/05/2014	14d
2	Implementación de simuladores SPI	19/05/2014	28/05/2014	10d
3	Diseño del bloque de captura de mensajes SPI (SPI_Sampler)	27/05/2014	04/06/2014	9d
4	Desarrollo de conversores de binario a BCD y de hexadecimal a ASCII	09/06/2014	16/06/2014	8d
5	Estudio y análisis del módulo UART	16/06/2014	20/06/2014	5d
6	Diseño del bloque de control del analizador SPI	16/06/2014	28/06/2014	13d
7	Implementación de memorias FIFO	01/07/2014	02/07/2014	2d
8	Pruebas de simulación y hardware del analizador SPI	02/07/2014	05/07/2014	4d
9	Diseño del bloque de captura de mensajes I2c	08/07/2014	20/07/2014	13d
10	Implementación del bloque de control del analizador I2C	23/07/2014	05/08/2014	14d
11	Pruebas del analizador I2C a nivel de simulación y hardware	11/08/2014	17/08/2014	7d
12	Implementación del bloque analizador de protocolos SPI e I2C	18/08/2014	21/08/2014	4d
13	Pruebas a nivel hardware del analizador de protocolos	21/08/2014	23/08/2014	3d
14	Documentación del proyecto	11/08/2014	07/09/2014	28d



7. Entorno socio-económico

Una vez que el diseño está listo para su uso, debe hacerse un análisis de los posibles entornos en los que se usará y el coste que supondrá si implementación.

7.1 Posibles entornos de aplicación

Considerando el amplio abanico de sectores en los que se emplean comunicaciones entre dispositivos, los posibles entornos en los que el diseño realizado puede emplearse son varios.

Principalmente, el diseño podría destinarse a los departamentos de desarrollo y verificación de diferentes industrias ya que será empleado para comprobar el correcto funcionamiento de otros dispositivos ya creados.

Dado que los buses SPI e I2C son empleados principalmente para comunicar a microcontroladores con sus periféricos, elementos empleados en numerosas industrias, el diseño puede ser utilizado en numerosos sectores. Industrias procedentes del sector automovilístico, de electrónica de consumo, de defensa o relacionadas con la salud son susceptibles de requerir el empleo de este diseño.

Por último, cabe destacar que sistema implementado puede ser muy útil a la hora de depurar el comportamiento de diversos prototipos, por lo que puede ser empleado en el campo de la investigación, tanto a nivel académico como a nivel industrial.

7.2 Presupuesto

El desarrollo de este proyecto supone una serie de costes que aparecen desglosados en el siguiente presupuesto:

Código	**Unidades**	**Descripción**	**Medición**	**Precio unitario (€)**	**Precio total**
CAPÍTULO 1: Desarrollo del diseño					
1.01	Ud	Licencia Xilinx ISE 13.4 del tipo Free ISE WebPack License	1	Gratuito	0
1.02	Ud	Ordenador personal con requisitos mínimos: - Sistema operativo Windows XP Pro SP3 o superior - Procesador de 1GHz o superior - Memoria RAM de 2 Gbytes como mínimo	1	680	680
1.03	Horas	Horas de ingeniería empleadas en el desarrollo. El diseño ha sido realizado por un único ingeniero.	600	18	10800

CAPÍTULO 2: validación del diseño					
2.01	Ud	Placa Xilinx Spartan 3E Starter Kit Board	1	154,45	154,45
2.02	Ud	Adaptador RS232_USB ICUSB232	1	14,56	14,56
2.03	m	Cable rígido unipolar de 0,5 mm de sección	2	0,51	1,02
2.04	Ud	Licencia de software PuTTY	1	Gratuito	0
2.05	Horas	Horas de ingeniería empleadas en la verificación.	70	18	1260
TOTAL:					**12910,03€**

8. Conclusiones

Una vez finalizadas las etapas de desarrollo del diseño e implementación de éste y la verificación de su funcionamiento pueden extraerse una serie de conclusiones:

- El diseño presentado es capaz de capturar tramas de datos de hasta 128 bits de longitud conectándose a un bus SPI como si de un esclavo más se tratase. Esto permite que el diseño se emplee para depurar otros elementos sin requerir ninguna modificación en el bus.
- Para cada uno de los mensajes SPI capturados el sistema identificará el número de bits transferidos, la polaridad del reloj de sincronización en reposo y el valor del parámetro de fase configurado. Esto permite identificar el modo de transmisión SPI empleado.
- Todos los mensajes capturados pueden ser visualizados en la pantalla de un ordenador empleando un software de terminal de equipo. La transmisión de los mensajes se realizará a través de un puerto serie RS-232.
- El sistema identifica cortes en las transmisiones SPI encendiendo un LED de notificación cuando la transmisión se ha parado sin finalizar. Si la transmisión se ha cortado con antelación el sistema lo identificará mediante la longitud de mensaje mostrada.
- El analizador de protocolos es capaz de capturar mensajes transferidos mediante el protocolo I2C independientemente de la longitud, expresada en bytes, del mensaje transmitido.
- El sistema identifica la dirección del esclavo al que se solicita la transmisión e indica si este ha sido correctamente direccionado. También es capaz de identificar el tipo de operación solicitada, sea de lectura o de escritura.
- Para cada uno de los bytes transmitidos se identifica el valor de la señal de asentimiento enviada por el receptor del mensaje.
- Todos los datos obtenidos pueden visualizarse en la pantalla de un ordenador presentando una estructura clara que permita su análisis posterior.
- El diseño puede implementarse aislado en una FPGA y conectarse físicamente al bus que se desee depurar. También puede emplearse como un módulo VHDL e incluirse en

proyecto de este tipo para depurar el sistema desde la propia FPGA en la que se implementa el sistema.

- La implementación realizada consume un número de recursos lógico bajo lo que hace que el sistema pueda incluirse junto con otros diseños en la misma FPGA sin comprometer la capacidad lógica de ésta.
- Existen varios campos de aplicación para este proyecto ya que las comunicaciones digitales son empleadas en diversos sistemas y todas requieren ser depuradas.

9. Líneas futuras

El sistema diseñado permite, gracias a su diseño jerárquico y modulado, que diferentes líneas de trabajo puedan llevarse a cabo en desarrollos posteriores:

- Implementación de funciones de detección de tiempo de espera (*timeout*) para cada uno de los analizadores implementados. Dicha función, que debería incluir un parámetro para configurar la velocidad de transmisión empleada, permitiría identificar cortes en la transferencia y velocidades de transmisión inadecuadas.
- Implementación de un detector de frecuencia de transmisión. Dicho elemento se encargaría de calcular el periodo del reloj de sincronización de cada bus y obtener la velocidad de transmisión empleada en cada conexión.
- Aumento de la velocidad de transmisión del puerto serie para disminuir el riesgo de pérdida de mensajes por falta de capacidad de las memorias empleadas.
- Contador de bytes para el analizador I2C. Así, se podría mostrar en pantalla el número de bytes de un mensaje y facilitar el análisis de los datos.
- Una nueva vía de trabajo sería el desarrollo de un software capaz de componer una representación temporal de los mensajes. Empleando la información transmitida por los analizadores, el sistema se encargaría de componer las gráficas temporales correspondientes a las transmisiones capturadas.
- Dado que el diseño ha sido dividido en módulos cuya función pude ser reutilizada, la implementación de un nuevo analizador para un protocolo distinto a los empleados hasta ahora resultaría sencilla y el tiempo de desarrollo sería menor al de este caso.
- Diseño de un circuito específico para el proyecto que incluya una FPGA y los LED e interruptores necesarios para el diseño.

10. Referencias y bibliografía

- [REF-1] Guía del bloque SPI de Motorola:

Autor/es:	©Motorola, Inc., 2001
Fecha de la última actualización:	4 de Febrero de 2003
Título:	*SPI Block Guide V03.06*
Fecha de consulta:	6 de Mayo de 2014
Dirección web:	http://www.ee.nmt.edu/~teare/ee308l/datasheets/S12SPIV3.pdf

- [REF-2] Especificación del bus I2C:

Autor/es:	©NXP Semiconductors
Fecha de la última actualización:	4 de Abril de 2014
Título:	*I2C-bus specification and user manual V.6*
Fecha de consulta:	14 de Mayo de 2014
Dirección web:	http://www.nxp.com/documents/user_manual/UM10204.pdf

- [REF-3] Interfaz entre equipos terminales y de comunicación de datos empleando intercambio de datos binarios en serie:

Autor/es:	*EIA(Electronics Industry Association)*
Fecha de edición:	1969
Título:	*EIA standard RS-232-C "Interface between data terminal equipment and data communication equipment employing serial binary data interchange*
Lugar de edición:	Estados Unidos

- [REF-4] Guía de usuario de la FPGA Spartan 3E Starter kit Board:

Autor/es:	©Xilinx
Fecha de la última actualización:	20 de Enero de 2011
Título:	*Spartan-3E FPGA Starter Kit Board User Guide*
Fecha de consulta:	6 de Agosto de 2014
Dirección web:	http://www.nxp.com/documents/user_manual/UM10204.pdf

Otra bibliografía consultada:

- Información general acerca del bus SPI:

Autor/es:	Wikipedia
Fecha de la última actualización:	30 de Agosto de 2013
Título:	*Serial Peripheral Interface*
Fecha de consulta:	5 de Mayo de 2014
Dirección web:	http://es.wikipedia.org/wiki/Serial_Peripheral_Interface

- Información general acerca del bus I2C:

Autor/es:	I2C.org
Fecha de la última actualización:	Sin fecha de edición
Título:	*I2C Bus*
Fecha de consulta:	12 de Mayo de 2014
Dirección web:	http://www.i2c-bus.org/

- Información del algoritmo recorre y suma 3:

Autor/es:	Wikipedia
Fecha de la última actualización:	22 de Agosto de 2014
Título:	*Double dabble*
Fecha de consulta:	25 de Agosto de 2014
Dirección web:	http://en.wikipedia.org/wiki/Double_dabble

- Web del fabricante Xilinx:

Autor/es:	©Xilinx
Fecha de la última actualización:	Sin fecha de edición
Título:	*All Programmable Technologies from Xilinx Inc.*
Fecha de consulta:	4 de Agosto de 2014
Dirección web:	http://www.xilinx.com/

11. Anexos

11.1 Características SPARTAN 3-E

Características de la FPGA SPARTAN -3E y funciones del procesador embebido

El kit de iniciación Spartan-3E destaca las características específicas de la familia de FPGAs Spartan-3E y ofrece un entorno de desarrollo adecuado para aplicaciones de procesamiento embebido. La placa resalta las siguientes características:

- Características específicas de la FPGA Spartan 3-E:
  - Configuración de NOR Flash paralela
  - Configuración de multiarranque de la FPGA desde NOR Flash PROM paralela
  - Configuración Flash mediante SPI serie
- Desarrollo embebido:
  - Procesador embebido RISC MicroBlaze™ 32-bit
  - Controlador embebido PicoBlaze™ 8-bit
  - Interfaz de memoria DDR

Placas de la generación Spartan 3-E de desarrollo avanzado

El kit de iniciación Spartan 3-E demuestra las capacidades básicas del procesador embebido MicroBlaze y el Kit de desarrollo embebido (EDK). Para desarrollos más avanzados considere las placas ofrecidas por los colaboradores de Xilinx:

http://www.xilinx.com/products/boards_kits/spartan.htm

Las características clave del Kit de iniciación Spartan3-E son:

- FPGA Spartan 3-E Xilinx XC3S500E
  - Hasta 232 pines de entrada/salida usables
  - Encapsulado FBGA de 320 pines
  - Más de 10.000 células lógicas
- PROM de configuración de plataforma Flash Xilinx de 4 Mbits
- CPLD Xilinx de 64 macroceldas XC2C64A CoolRunner™
- 64 MBytes (512 Mbits) de memoria DDR SDRAM, 16 interfaces de datos, + de 100 MHz
- 16 MBytes (128 Mbits) de NOR Flash (Intel StrataFlash) paralela
  - Almacenamiento de configuración de FPGA
  - Almacenamiento/duplicación de código del MicroBlaze
- 16 MBits de Flash controlada por SPI serie (STMicro)
  - Almacenamiento de configuración de FPGA
  - Duplicación de código del MicroBlaze
- Pantalla LCD de 16 caracteres y 2 líneas
- Puerto PS/2 de ratón o teclado
- Puerto de pantalla VGA

- Ethernet 10/100 PHY (requiere MAC Ethernet en la FPGA)
- Dos puertos de 9 pines RS-232 (estilo DTE y DCE)
- Puerto USB para depuración/programación de FPGA/CPLD
- Oscilador de reloj de 50 MHz
- EEPROM serie de un hilo SHA-1 para protección de copia del bitstream
- Conector de expansión Hirose FX2
- Tres conectores de expansión Digilent de 6 pines
- Conversor de Digital a Analógico (DAC) basado en SPI de 4 salidas
- Conversor de Analógico a Digital (ADC) basado en SPI de 2 entradas con preamplificador de ganancia programable
- Puerto para depuración ChipScope™ SoftTouch
- Encoder rotatorio con eje pulsador
- Ocho LEDs discretos
- Cuatro interruptores desplazables
- Cuatro pulsadores
- Entrada de reloj SMA
- Zócalo de 8 pines con encapsulado DIP para oscilador de reloj auxiliar

Compromisos de diseño

Algunos compromisos de diseño a nivel de sistema son requeridos para proporcional al kit de iniciación Spartan 3-E la mayor funcionalidad

¡Abundantes métodos de configuración!

Una aplicación típica de la FPGA usa una única memoria no volátil para almacenar imágenes de configuración. Para demostrar las capacidades de la nueva Spartan 3-E, el kit de iniciación Spartan 3-E tiene tres fuentes de configuración de memoria que necesitan funcionar correctamente juntas. Las funciones extra de configuración hacen al kit de iniciación Spartan 3-E más complejo que las aplicaciones típicas de las FPGA Spartan 3-E.

La placa del kit de iniciación incluye además una interfaz de programación JTAG basada en un USB que incluye la placa. La circuitería sobre el chip simplifica la experiencia de programación del dispositivo. En aplicaciones comunes, el hardware de programación JTAG se encuentra fuera de la placa o en un módulo de programación, como el cable Xilinx Platform USB.

Tensiones para todas las aplicaciones

El Kit de iniciación Spartan 3-E presenta un regulador de triple salida desarrollado por Texas Instruments, el TPS75003 específicamente diseñado para alimentar FPGAs Spartan e y Spartan 3-E. Este regulador es suficiente para la mayoría de aplicaciones donde la FPGA trabaja sola. No obstante, la placa del Kit de iniciación incluye memoria DDR

SDRAM, la cual requiere su propia fuente de alta corriente. De forma similar, el JTAG de descarga basado en USB requiere una fuente separada de 1,8 V.

11.2 Fichero de restricciones del diseño para Spartan 3E

```
##############################################################
### SPARTAN-3E STARTER KIT BOARD CONSTRAINTS FILE
##############################################################
# ==== Discrete LEDs (LED) ====
NET "LED_SPI_EMPTY" LOC = "F9" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LED_SPI_MSG_LOST" LOC = "E9" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LED_SPI_RUNNING" LOC = "D11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LED_I2C_EMPTY" LOC = "C11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LED_I2C_MSG_LOST" LOC = "F11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
NET "LED_I2C_RUNNING" LOC = "E11" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 8 ;
# ==== Slide Switches (SW) ====
NET "RESET" LOC = "N17" | IOSTANDARD = LVTTL | PULLUP ;
NET "SELECT_PROTOCOL" LOC = "H18" | IOSTANDARD = LVTTL | PULLUP ;

# ==== Clock inputs (CLK) ====
NET "CLK" LOC = "C9" | IOSTANDARD = LVCMOS33 ;
# Define clock period for 50 MHz oscillator (40%/60% duty-cycle)
NET "CLK" PERIOD = 20.0ns HIGH 50%;

# ==== RS-232 Serial Ports (RS232) ====
NET "RS232_TX_PAD" LOC = "M14" | IOSTANDARD = LVTTL | DRIVE = 8 | SLEW = SLOW ;

# ==== 6-pin header J1 ====
# These four connections are shared with the FX2 connector
NET "MOSI_IN" LOC = "B4" | IOSTANDARD = LVTTL | PULLUP | SLEW = SLOW | DRIVE = 6 ;
NET "MISO_IN" LOC = "A4" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 6 ;
NET "SCLK_IN" LOC = "D5" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 6 ;
NET "CS_IN" LOC = "C5" | IOSTANDARD = LVTTL | PULLUP | SLEW = SLOW | DRIVE = 6 ;
# ==== 6-pin header J2 ====
# These four connections are shared with the FX2 connector
NET "SDA_IN" LOC = "A6" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 6 ;
NET "SCL_IN" LOC = "B6" | IOSTANDARD = LVTTL | SLEW = SLOW | DRIVE = 6 ;
```

11.3 Ficha técnica del adaptador de USB a RS-232 ICUSB232

Cable de 0,3m USB a Puerto Serie Serial RS232 DB9

StarTech ID: ICUSB232

El Cable Adaptador USB a Puerto Serie RS232 DB9 de 1 pie de Startech.com permite conectar dispositivos seriales RS232 DB9 a su Mac o PC portátil o de escritorio a través de un puerto USB, como si el ordenador tuviera un conector DB9M incorporado.

Una solución eficaz y económica que permite acortar la distancia en términos de compatibilidad entre ordenadores modernos y periféricos preexistentes con conectividad serial.

Viene con 2-años de garantía y soporte técnico gratuito de por vida con el respaldo de StarTech.com

Aplicaciones

- Perfecto para Administradores IT que buscan incorporar funcionalidades existentes a portátiles, PCs y servidores más modernos desprovistos de puertos RS232 integrados
- Permite conectar un receptor satelital, modem serial, o PDA con sync serial
- Permite conectar escáneres de código de barras, impresoras de facturas y otros dispositivos de puntos de venta
- Permite conectar, monitorear y controlar sensores industriales/automotrices y equipos
- Permite conectar y programar tableros de señalización LED y Digital equipados con puertos de comunicación serial
- Compatible con la mayoría de las marcas que fabrican PLCs (AllenBradley, Siemens, Modicon, Servos, Indramat, Siemens, AB, etc.)

Características

Adaptador Directo USB a Serie RS232

Funciones fáciles de configurar y de instalar

No requiere alimentación externa – se alimenta a través de la conexión USB Al ser cables delgados y livianos son fáciles de transportar

Soporta velocidades de transferencia de datos de hasta 1 Mbits/seg

Compatible con los PDAs, módems, impresoras, escáners de código de barras, etc. más comunes.

Especificaciones Técnicas

Cantidad de Puertos	1
Interfaz	Serial
Tipo de Bus	USB 2.0
Estilo de Puerto	Cable Adaptador
Estándares Industriales	USB 1.1/2.0
	RS232
ID del Conjunto de Chips	Prolific - PL-2303 RS-232
Tipo(s) de Conector(es)	
	1 - DB-9 (9 pin; D-Sub) Macho Tipo(s) de Conector(es) 1 - USB A (4 pin) Macho Protocolo Serie
Tasa Máxima de Baudios	921,6 Kbps
FIFO	192 Bytes
Compatibilidad OS	Windows® 7 (32/64bit), Vista (32/64), XP (32/64), 2000, ME, 98SE, CE 4.2, 5.2
	Windows® Server 2012, 2008 R2, 2003(32/64) Mac OS® 10.x
Tipo de Gabinete	Plástico
Longitud del Cable	304.8 mm [12 in]
Longitud del Producto	370 mm [14.6 in]

11.4 Código fuente (VHDL) de los principales bloques del diseño

11.4.1 Analizador de protocolos

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity TOP_SPI_I2C_ANALYZER is
   Port ( CLK : in  STD_LOGIC;
          RESET : in  STD_LOGIC;
          SELECT_PROTOCOL : in  STD_LOGIC;
          MISO_IN : in  STD_LOGIC;
          MOSI_IN : in  STD_LOGIC;
          SCLK_IN : in  STD_LOGIC;
          CS_IN : in  STD_LOGIC;
          SDA_IN : in  STD_LOGIC;
          SCL_IN : in  STD_LOGIC;
          RS232_TX_PAD : out  STD_LOGIC;
          LED_SPI_EMPTY : out  STD_LOGIC;
          LED_SPI_MSG_LOST : out  STD_LOGIC;
          LED_SPI_RUNNING : out  STD_LOGIC;
          LED_I2C_EMPTY : out  STD_LOGIC;
          LED_I2C_MSG_LOST : out  STD_LOGIC;
          LED_I2C_RUNNING : out  STD_LOGIC);
end TOP_SPI_I2C_ANALYZER;
architecture Behavioral of TOP_SPI_I2C_ANALYZER is

COMPONENT TOP_SPI
   Port ( CLK : in  STD_LOGIC;
          RESET : in  STD_LOGIC;
          MISO_I : in  STD_LOGIC;
          MOSI_I : in  STD_LOGIC;
          CS_I : in  STD_LOGIC;
          SCLK_I : in  STD_LOGIC;
-- Output connected to a LED to recognize that the FIFO memories are empty. It means that there are no messages
to send
          LED_EMPTY: out  STD_LOGIC;
-- Output connected to a LED to indicate that any message has been lost because when it was received the FIFO
memories were full.
          LED_MSG_LOST : out  STD_LOGIC;
          LED_RUNNING : out  STD_LOGIC;             -- Output to indicate that the module is reveiving a message.
          TX_PAD_O : out  STD_LOGIC);
end COMPONENT;

COMPONENT TOP_I2C
   Port ( CLK : in  STD_LOGIC;
          RESET : in  STD_LOGIC;
          SDA_in : in  STD_LOGIC;
          SCL_in : in  STD_LOGIC;
          LED_EMPTY : out  STD_LOGIC;
          LED_MSG_LOST : out  STD_LOGIC;
          LED_RUNNING : out  STD_LOGIC;
          TX_pad : out  STD_LOGIC);
end COMPONENT;
-- Signals
signal reset_SPI: std_logic; -- signal to stop the SPI analyzer wwhen the I2C analyzer is selected
signal reset_I2C: std_logic; -- signal to stop the I2C analyzer wwhen the SPI analyzer is selected
signal RS232_SPI: std_logic; -- serial output of the SPI analyzer
signal RS232_I2C: std_logic; -- serial output of the I2C analyzer

begin

-- INSTANTIATIONS
```

```
SPI_Analyzer: TOP_SPI
   Port map ( CLK => CLK,
         RESET => reset_SPI,
         MISO_I => MISO_IN,
         MOSI_I => MOSI_IN,
         CS_I => CS_IN,
         SCLK_I => SCLK_IN,
         LED_EMPTY => LED_SPI_EMPTY,
         LED_MSG_LOST => LED_SPI_MSG_LOST,
         LED_RUNNING => LED_SPI_RUNNING,
         TX_PAD_O => RS232_SPI
);

I2C_Analyzer: TOP_I2C
   Port map ( CLK => CLK,
         RESET => reset_I2C,
         SDA_in => SDA_IN,
         SCL_in => SCL_IN,
         LED_EMPTY => LED_I2C_EMPTY,
         LED_MSG_LOST => LED_I2C_MSG_LOST,
         LED_RUNNING => LED_I2C_RUNNING,
         TX_pad => RS232_I2C
);

-- Reset management
reset_SPI <= RESET or SELECT_PROTOCOL;
reset_I2C <= RESET or not (SELECT_PROTOCOL);
-- RS232 multiplexer
RS232_TX_PAD <= RS232_SPI when SELECT_PROTOCOL = '0' else RS232_I2C;

end Behavioral;
```

11.4.2 Top SPI

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

entity TOP_SPI is
   Port ( CLK : in  STD_LOGIC;
         RESET : in  STD_LOGIC;
         MISO_I : in  STD_LOGIC;
         MOSI_I : in  STD_LOGIC;
         CS_I : in  STD_LOGIC;
         SCLK_I : in  STD_LOGIC;
-- Output connected to a LED to recognize that the FIFO memories are empty. It means that there are no messages
to send
         LED_EMPTY: out  STD_LOGIC;
-- Output connected to a LED to indicate that any message has been lost because when it was received the FIFO
memories were full.
         LED_MSG_LOST : out  STD_LOGIC;
         LED_RUNNING : out  STD_LOGIC;              -- Output to indicate that the module is receiving a message.
         TX_PAD_O : out  STD_LOGIC);
end TOP_SPI;

architecture Behavioral of TOP_SPI is
-- COMPONENTS
COMPONENT SPI_Sampler
 Generic (-- This generic defines the maximum number of bits that a message could contain.
            G_MAX_NUMBER_OF_BITS     : natural);
   Port ( CLK : in  STD_LOGIC;
         RESET : in  STD_LOGIC;
```

```
        DATA_MOSI : out  STD_LOGIC_VECTOR ((G_MAX_NUMBER_OF_BITS - 1) downto 0);
        DATA_MISO : out  STD_LOGIC_VECTOR ((G_MAX_NUMBER_OF_BITS - 1) downto 0);
        NUM_BITS : out  STD_LOGIC_VECTOR (7 downto 0);
        PHASE : out std_logic;
        POLARITY : out std_logic;
        NEW_MSG: out std_logic;
        MOSI : in  STD_LOGIC;
        MISO : in  STD_LOGIC;
        SCLK : in  STD_LOGIC;
        CS : in  STD_LOGIC;
        RUNNING : out  STD_LOGIC);
END COMPONENT;

COMPONENT CONTROL_BLOCK
   PORT(
       CLK : IN  std_logic;
       RESET : IN  std_logic;
       NEW_DATA : IN  std_logic;
       TX_PAD : OUT  std_logic;
       LED_EMPTY : OUT  std_logic;
       LED_MSG_LOST: OUT  std_logic;
       FIFO_16_IN : IN  std_logic_vector(15 downto 0);
       FIFO_256_IN : IN  std_logic_vector(255 downto 0)
      );
   END COMPONENT;

-- TYPES
type array_data is array (0 to 9) of std_logic_vector(9 downto 0);
-- SIGNALS
-- SAMPLER CONECTORS
signal master_slave_data: std_logic_vector (255 downto 0); -- complete message which will be stored in the 256 b
FIFO memory
signal counter_and_params: std_logic_vector (15 downto 0); -- number of bits, polarity and phase of each message
which will be stored in the 16 b FIFO memory
signal master_msg: std_logic_vector (127 downto 0); -- Master message
signal slave_msg: std_logic_vector (127 downto 0); -- Slave message
signal n_bits_msg: std_logic_vector (7 downto 0); -- Nuomber of bits of message
signal cpol_value: std_logic; -- 1 bit polarity
signal cpha_value: std_logic; --  1 bit phase
signal new_msg: std_logic; -- Signal to indicate that a new message has been captured

begin

Sampler_uut: SPI_Sampler
GENERIC MAP(
        G_MAX_NUMBER_OF_BITS => 128)
PORT MAP ( CLK => CLK,
        RESET => RESET,
        DATA_MOSI => master_msg,
        DATA_MISO => slave_msg,
        NUM_BITS => n_bits_msg,
        PHASE => cpha_value,
        POLARITY => cpol_value,
        NEW_MSG => new_msg,
        MOSI => MOSI_I,
        MISO => MISO_I,
        SCLK => SCLK_I,
        CS => CS_I,
        RUNNING => LED_RUNNING
);

control_uut: CONTROL_BLOCK PORT MAP (
```

```
        CLK => CLK,
        RESET => RESET,
        NEW_DATA => new_msg,
        TX_PAD => TX_PAD_O,
        LED_EMPTY => LED_EMPTY,
        LED_MSG_LOST => LED_MSG_LOST,
        FIFO_16_IN => counter_and_params,
        FIFO_256_IN => master_slave_data
);

-- OUTPUTS ASSIGNMENT
master_slave_data <= master_msg & slave_msg;
counter_and_params <= n_bits_msg & "000" & cpol_value & "000" & cpha_value;

end Behavioral;
```

11.4.3 Bloque SPI_Sampler

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_arith.ALL;
use ieee.math_real.all;

entity SPI_Sampler is
            Generic ( G_MAX_NUMBER_OF_BITS     : natural); -- This generic defines the maximum number of bits
that a message could contain.
   Port ( CLK : in  STD_LOGIC;               -- Main clock
        RESET : in  STD_LOGIC;               -- Reset actived high port
        DATA_MOSI : out  STD_LOGIC_VECTOR ((G_MAX_NUMBER_OF_BITS - 1) downto 0);          -- Output which
contains the message captured from the MOSI line
        DATA_MISO : out  STD_LOGIC_VECTOR ((G_MAX_NUMBER_OF_BITS - 1) downto 0);          -- Output which
shows the information captured from the MISO line
        NUM_BITS : out  STD_LOGIC_VECTOR (7 downto 0);               -- Number of bits of the captured message
        PHASE : out std_logic;               -- Phase configuration of the SPI transmission mode detected in a message
        POLARITY : out std_logic;            -- Polarity configuration of the SPI transmission mode detected in a message
        NEW_MSG: out std_logic;              -- Output to indicate that a new message has been captured
        MOSI : in  STD_LOGIC;        -- Master Output Slave Input. Serial line to capture the message sent by the Master
        MISO : in  STD_LOGIC;        -- Master Input Slave Output. Serial input to capture the message sent by the Slave
        SCLK : in  STD_LOGIC;        -- Serial Clock input to siynchronize the reception of the message
        CS : in  STD_LOGIC;-- Chip Select. Input to identify that the slave who will be analyzed is selected
RUNNING : out  STD_LOGIC);       -- Output to indicate that the module is reveiving a message.
end SPI_Sampler;

architecture Behavioral of SPI_Sampler is
-- Constants
constant C_EDGES_COUNTER_WD       : natural:= natural(ceil(log2(real((G_MAX_NUMBER_OF_BITS*2)+ 1)))); --
Constant to define the width of the SCLK edges counter
-- FSM signals
type state is (idle,transmission);           -- FSM states
signal r_state : state; -- Actual state
signal nr_state: state; -- next state

-- Bits counter signals
signal r_edges_counter: unsigned((C_EDGES_COUNTER_WD - 1) downto 0); -- SCLK edges counter to identify the
number of bits received
signal reset_edge_count: std_logic; -- Signal to set to zero the counter for a new transmission -- SCLK signals
signal sclk_signal: std_logic; -- SCLK input register
signal r_sclk_signal: std_logic;             -- Second SCLK input register to avoid metastability
signal r2_sclk_signal: std_logic;
signal r3_sclk_signal: std_logic;
signal sclk_edge: std_logic; -- Signal to identify an edge in the SCLK line
-- Shift registers to store the data received from MOSI input (Master message) and from MOSI input (Slave message)
```

```
signal r_shift_data: std_logic_vector(G_MAX_NUMBER_OF_BITS - 1 downto 0);
signal r_shift_data_miso: std_logic_vector(G_MAX_NUMBER_OF_BITS - 1 downto 0);
-- Serial input registers
signal sda_in: std_logic; -- Register of the MOSI input
signal miso_in: std_logic; -- Register of the MISO input
signal r_chip_select: std_logic; -- Register of Chip Select input
-- Control signals
signal pha_lock: std_logic;             -- Signal that indicates that the phase parameter has been identified
signal enable_rx: std_logic;            -- Signal to read a new bit from the serial inputs
signal edge_to_rx: std_logic; -- Signal which indicates if a new bit will be read with rising or falling edge of SCLK
signal end_trans: std_logic;            -- End of transmission
signal enable_tx: std_logic;            -- Signal to enable the edge detection and the serial inputs analysis
signal flag_first_0_a: std_logic;       -- Flag to identify that the first zero has been identified in the MOSI input.
signal new_data: std_logic;             -- A new read operation could be done
signal r_new_data: std_logic;           -- register of new data signal
signal first_0_a: std_logic;            -- The first zero value has been identified in the MOSI serial line
signal flag_first_0_b: std_logic;-- Flag to identify that the first zero has been identified in the MISO input.
signal first_0_b: std_logic;            -- The first zero value has been identified in the MISO serial line
signal s_running: std_logic;            -- Signal to identify that the module is analyzing a message

begin

register_process:process (RESET,CLK)
begin
if RESET= '1' then
                r_state <= idle;
                sclk_signal <=      '0';
                r_sclk_signal <= '0';
                r2_sclk_signal <= '0';
                r3_sclk_signal <= '0';
                sclk_edge <= '0';
                DATA_MOSI <=    (others=>'0');
                DATA_MISO <=    (others=>'0');
                NUM_BITS <=     (others=>'0');
                PHASE <= '0';
                POLARITY <= '0';
                r_edges_counter <= (others=>'0');
                r_chip_select <= '1';
                r_new_data <= '0';
elsif CLK'event and clk = '1' then
                r_state <= nr_state;
                sclk_signal <=      SCLK;
                r_sclk_signal <=    sclk_signal;
                r2_sclk_signal <= r_sclk_signal;
                r3_sclk_signal <= r2_sclk_signal;
                r_chip_select <= CS;
                r_new_data <= new_data;
-- SCLK Edge detector. This operation is allowed only when a new message is being received
-- This condition avoid the detection of an edge when the polarity of the SCLK line changes between messages.
                if ((r3_sclk_signal xor r2_sclk_signal) = '1') and enable_tx = '1' then
                        sclk_edge <= '1';
                else
                        sclk_edge <= '0';
                end if;
                if  reset_edge_count = '1' then -- Set to zero the edges counter for a new message
                        r_edges_counter <= (others=>'0');
-- when an edge is detected the counter is increased
                elsif (r3_sclk_signal xor r2_sclk_signal) = '1' then
                        r_edges_counter <= r_edges_counter + 1;
                else
                        r_edges_counter <= r_edges_counter;
                end if;
```

```
-- when the end of a message is identified outputs show the message parameters
                        if end_trans = '1' then
                          DATA_MOSI <= r_shift_data;
                          DATA_MISO <=   r_shift_data_miso;
                          NUM_BITS <= conv_std_logic_vector (r_edges_counter((r_edges_counter'high) downto 1), 8);
                          PHASE <= not edge_to_rx;
                          POLARITY <= r3_sclk_signal;
                        end if;
            end if;
end if;
end process;

shift_data_register_process:process (RESET,CLK)
begin
if RESET= '1' then
            r_shift_data <= (others=>'1');
            r_shift_data_miso <= (others=>'1');
elsif CLK'event and clk = '1' then
-- Shift registers are set to 1 (idle state of serial inputs) when a transmission has finished
            if end_trans = '1' then
                        r_shift_data <= (others=>'1');
                        r_shift_data_miso <= (others=>'1');
-- When a new read operation is requested, the serial inputs values are stored in the shift registers
            elsif enable_rx = '1' then
                        r_shift_data <= r_shift_data((r_shift_data'length-2) downto 0) & sda_in; -- MOSI messages
                        r_shift_data_miso <= r_shift_data_miso((r_shift_data_miso'length-2) downto 0) & miso_in; --
MISO messages
            end if;
end if;
end process;

MOSI_and_MISO_Analyze_process:process (RESET,CLK) -- Process to identify the phase parameter of a message
begin
if RESET= '1' then
            pha_lock <= '0';
edge_to_rx <= '0';
            flag_first_0_a <= '0';
            first_0_a <= '0';
            flag_first_0_b <= '0';
            first_0_b <= '0';
elsif CLK'event and clk = '1' then
            if end_trans = '1' then -- When a transmission ends all signals are set to zero
                        pha_lock <= '0';
                        edge_to_rx <= '0';
                        flag_first_0_a <= '0';
                        first_0_a <= '0';
                        flag_first_0_a <= '0';
                        first_0_b <= '0';
            elsif new_data = '1' then -- A new read operation is allowed
                        if sda_in = '0' and first_0_a = '0' and pha_lock = '0'  then -- first 0 received in the MOSI line
-- First edge. We couldn´t know if the cero value was placed in the serial line before the first edge arrives or after it,
so we couldn´t identify yet the phase of the message
                                    if r_edges_counter = 1 then
                                                edge_to_rx <= r_edges_counter(0); -- We will read serial lines with odd edges
                                                first_0_a <= '1'; -- First zero has been read
                                                flag_first_0_a <= '1';
-- If the first zero is identified in an edge that isn´t the first edge received, we can identify the phase of the message
because we know that the serial value has changed in this edge
                                    else
                                                pha_lock <= '1'; -- Phase has been identified
                                                edge_to_rx <= not (r_edges_counter(0)); -- We will read next bits with edges
where data is stable, not when they changes that is the identified edge
```

```
                        end if;
-- first 1 received after the first 0. We could identify the phase of the message.
-- We will read next bits with edges where data is stable, not when they changes that is the identified edge
                elsif sda_in = '1' and first_0_a = '1'  then
                        pha_lock <= '1';
                        edge_to_rx <= not (r_edges_counter(0));
                        first_0_a <= '0';
                end if;
                if miso_in = '0' and first_0_b = '0' and pha_lock = '0'  then -- first 0 received. Similar to MOSI
                        if r_edges_counter = 1 then -- first edge, phase couldn´t be identified yet
                                edge_to_rx <= r_edges_counter(0);
                                first_0_b <= '1';
                                flag_first_0_b <= '1';
                        else -- Other edge, phase can be identified
-- We will read next bits with edges where data is stable, not when they changes that is the identified edge
                                pha_lock <= '1';
                                edge_to_rx <= not (r_edges_counter(0));
                        end if;
                elsif miso_in = '1' and first_0_b = '1'  then -- first 1 received after the first 0. Phase is identified
-- We will read next bits with edges where data is stable, not when they changes that is the identified edge
                        pha_lock <= '1';
                        edge_to_rx <= not (r_edges_counter(0));
                        first_0_b <= '0';
                        end if;
                else
                        flag_first_0_a <= '0';
                        flag_first_0_b <= '0';
                end if;
end if;
end process;

enable_rx_process:process (RESET,CLK) -- Process to enable the serial lines read operation
begin
if RESET= '1' then
enable_rx <= '0';
elsif CLK'event and clk = '1' then
-- A new bit is received and the phase is locked. We read the serial lines when the data is stable
        if r_new_data = '1' and pha_lock = '1' and (edge_to_rx = r_edges_counter(0)) then
        enable_rx <= '1';
-- The first zero was identified but the phase isn´t recognized yet. We read the serial lines each couple of edges
        elsif r_new_data = '1' and (first_0_a = '1' or first_0_b = '1') and (edge_to_rx = r_edges_counter(0)) then
        enable_rx <= '1';
        elsif flag_first_0_a = '1' or flag_first_0_b = '1' then -- If a first zero is detected, it´s stored in the register
                enable_rx <= '1';
        else -- If any of the previous conditions isn´t identified, the read operation is disabled.
                enable_rx <= '0';
        end if;
end if;
end process;

inputs_reading_process:process (RESET,CLK) -- Signal to register the serial lines value with each SCLK edge
begin
if RESET= '1' then
        sda_in <= '1';
        miso_in <= '1';
        new_data <= '0';
elsif CLK'event and clk = '1' then
-- When the operation is allowed we read the serial lines. This condition avoid the read operation when the SCLK
line changes due to the polarity configuration between messages
        if enable_tx = '1' then
                if sclk_edge = '1' then
                        sda_in <= MOSI;
```

```
                        miso_in <= MISO;
                        new_data <= '1';
                else
                        new_data <= '0';
                end if;
        else -- Idle values
                sda_in <= '1';
                miso_in <= '1';
                new_data <= '0';
        end if;
end if;
end process;

fsm_process: process (r_chip_select, CS, r_state)
begin
-- Default values
nr_state <= r_state;
        end_trans <= '0';
        reset_edge_count <= '0';
        enable_tx <= '0';
        s_running <= '0';
case r_state is
when idle =>
                reset_edge_count <= '1';
                end_trans <= '0';
                enable_tx <= '0'; -- Other components are on idle state
                s_running <= '0'; -- The analyzer is stopped
-- A falling edge is detected in the Chip Select lines so a new transmission begins
                if r_chip_select = '1' and CS = '0' then
                        nr_state <= transmission;
                        enable_tx <= '1';
                end if;

        when transmission =>
                enable_tx <= '1'; -- Other compents are enabled
                s_running <= '1'; -- the analyzer is receiving data
-- A rising edge is detected in the Chip Select lines so the message has been sent completelly
                if r_chip_select = '0' and CS = '1' then
                        nr_state <= idle;
                        end_trans <= '1';
                end if;
        when others => nr_state <= idle;
 end case;
end process;
-- A new message has been completely received when a transmission ends
NEW_MSG <= '0' when RESET = '1' else end_trans;
RUNNING <= s_running;
end Behavioral;
```

11.4.4 Bloque de control SPI

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
entity CONTROL_BLOCK is
   Port ( CLK : in  STD_LOGIC;              -- Main clock port
          RESET : in  STD_LOGIC;            -- Reset actived high port
          NEW_DATA : in  STD_LOGIC;         -- Input from the sampler which indicates that a message has been received
          TX_PAD : out  STD_LOGIC;          -- RS-232 serial output to comunicate with the PC
-- Output connected to a LED to recognize that the FIFO memories are empty. It means that there are no messages
to send
          LED_EMPTY : OUT  std_logic;
```

```
-- Output connected to a LED to indicate that any message has been lost because when it was received the FIFO
memories were full.
        LED_MSG_LOST : out  STD_LOGIC;
-- FIFO 16 bits data input (number of bits, polarity and phase of a message)
        FIFO_16_IN : in  STD_LOGIC_VECTOR (15 downto 0);
-- FIFO 256 bits data input (master message and slave message)
        FIFO_256_IN : in  STD_LOGIC_VECTOR (255 downto 0));
end CONTROL_BLOCK;

architecture Behavioral of CONTROL_BLOCK is
-- COMPONENTS
COMPONENT FIFO_16B -- 16 bits FIFO memory with depth = 1024 messages
  PORT (
    clk : IN STD_LOGIC;           -- Main clock port
    rst : IN STD_LOGIC;           -- Reset actived high port
    din : IN STD_LOGIC_VECTOR(15 DOWNTO 0);  -- FIFO data input
    wr_en : IN STD_LOGIC;                    -- Write enable input
    rd_en : IN STD_LOGIC;                    -- Read eable input
    dout : OUT STD_LOGIC_VECTOR(15 DOWNTO 0);-- FIFO data output
    full : OUT STD_LOGIC;                    -- Output to indicate that the memory is full
    wr_ack : OUT STD_LOGIC;                  -- Write acknowledge
    empty : OUT STD_LOGIC;                   -- Output to indicate that the memory is full
    valid : OUT STD_LOGIC  );
END COMPONENT;

COMPONENT FIFO_256B -- 256 bits FIFO memory with depth = 1024 messages
  PORT (    clk : IN STD_LOGIC;
    rst : IN STD_LOGIC;
    din : IN STD_LOGIC_VECTOR(255 DOWNTO 0);
    wr_en : IN STD_LOGIC;
    rd_en : IN STD_LOGIC;
    dout : OUT STD_LOGIC_VECTOR(255 DOWNTO 0);
    full : OUT STD_LOGIC;
    wr_ack : OUT STD_LOGIC;
    empty : OUT STD_LOGIC;
    valid : OUT STD_LOGIC
  );
END COMPONENT;

-- This component converts a 4 bits vector(hexadecimal bit) to its correspondant ASCII value
COMPONENT HEX_TO_ASCII_CONVERTER     PORT ( CLK : in  STD_LOGIC;
        RST : in  STD_LOGIC;
        DIN : in  STD_LOGIC_VECTOR (3 downto 0);
        ENABLE : in  STD_LOGIC;
        DOUT : out  STD_LOGIC_VECTOR (7 downto 0);
        DOUT_VALID : out  STD_LOGIC);
end COMPONENT;

COMPONENT BIN_TO_DEC_CONVERTER -- This component receives a binary data and returns its value in bcd
    PORT ( CLK : in  STD_LOGIC;
        RST : in  STD_LOGIC;
        DIN : in  STD_LOGIC_VECTOR (7 downto 0);
        ENABLE : in  STD_LOGIC;
        DOUT : out  STD_LOGIC_VECTOR (11 downto 0);
        DOUT_VALID : out  STD_LOGIC);
end COMPONENT;

COMPONENT UART -- RS_232 transmitter
  generic(BRDIVISOR: INTEGER range 0 to 65535 );--130); -- Baud rate divisor
  port (
     BR_Clk_I : in  std_logic;  -- Clock used for Transmit/Receive
     reset    : in  std_logic;
```

```
    TxD_PAD_O: out std_logic;  -- Tx RS232 Line
    RxD_PAD_I: in  std_logic;  -- Rx RS232 Line
    byteFromPC: out std_logic_vector(7 downto 0);
    LoadTx    : in  std_logic;
    Byte2PC   : in std_logic_vector(7 downto 0);
    TxBusy    : out std_logic;  -- Transmiter Busy
    RxAv      : out std_logic  -- Data Received
    );
end COMPONENT;

-- TYPES
type state is (idle, send_cab, send_bcd, conv_ASCII, uart_and_control, shift_128b, message_Data,
send_pol_and_phase);          -- FSM states
-- SIGNALS
-- FSM signals
signal r_state : state; -- Actual state signal
signal nr_state: state; -- Next state signal
signal r_control : std_logic_vector (3 downto 0); -- signal to control transitions
signal nr_control : std_logic_vector (3 downto 0);
-- Counters
signal pos_cab: unsigned (3 downto 0); -- Counter to send the correct header character
signal add_pos: std_logic; -- signal to increment the position counter
signal rst_pos: std_logic; -- signal to reset the position counter
signal aux_cnt: unsigned (7 downto 0); -- Auxiliar couner used for sending the master and slave messages
signal load_128: std_logic; -- signal to load the value to the counter
signal load_2: std_logic; -- signal to load the value to the counter
signal substract_1: std_logic; -- signal to decrement 1 the counter value
signal substract_4: std_logic; -- signal to decrement 4 the counter value
signal msg_cnt: unsigned (7 downto 0); -- Counter of messages sent (limit = 256)
signal add_msg: std_logic; -- signal to increment the position counter
-- Registers
-- 128 bits
signal r_128b: std_logic_vector (127 downto 0); -- register used for load the master and slave data
signal load_master: std_logic; -- signal to load the master data
signal load_slave: std_logic; -- signal to load the slave data
signal shift_4: std_logic; -- signal to shift the register 4 positions
signal shift_1: std_logic; -- signal to add one zero to the 128 bits register
signal shift_2: std_logic; -- signal to add two zero to the 128 bits register
signal shift_3: std_logic; -- signal to add three zero to the 128 bits register
-- 12 bits
signal r_12b: std_logic_vector (11 downto 0); -- register used for load bcd results
signal shift_bcd: std_logic; -- signal to shift the register 4 positions
-- BCD conversor
signal r_bcd_o: std_logic_vector (11 downto 0); -- register used for load bcd results
signal enable_bcd: std_logic; -- signal to enable the binary to decimal converter
signal bcd_valid: std_logic; -- signal to load the binary to decimal result
signal r_bcd_valid: std_logic; -- signal to load the binary to decimal result
signal r2_bcd_valid: std_logic; -- signal to load the binary to decimal result
-- ASCII conversor
signal r_data2conv: std_logic_vector (3 downto 0); -- register used for load a 4 bits gropu to convert
signal nr_data2conv: std_logic_vector (3 downto 0); -- register used for load a 4 bits gropu to convert
signal r_data_conv_o: std_logic_vector(7 downto 0); -- 8 bits vector that contains data converted
signal enable_conv: std_logic; -- signal to enable the hex to ASCII converter
signal conv_valid: std_logic; -- signal to load the hex to ASCII result
-- UART
signal r_data2tx: std_logic_vector(7 downto 0); -- 8 bits vector that contains data to send
signal nr_data2tx: std_logic_vector(7 downto 0);
signal load_tx: std_logic; -- This signal indicates that a new transmission is required
signal tx_busy: std_logic; -- Signal which indicates that the UART is busy
signal r_tx_end: std_logic; -- Signal which indicates that the message is sent and a carriage return must be sent
signal nr_tx_end: std_logic;
--FIFOs
```

```
signal wr_en: std_logic; -- Write enable signal of FIFO memory
signal rd_en: std_logic; -- Read enable signal of FIFO memory
signal FIFO_dout_256: std_logic_vector(255 downto 0); -- 272 bits vector that contains a message stored in FIFO
signal full_256: std_logic; -- signal to indicate that the FIFO memory is full
signal wr_ack_256: std_logic; -- FIFO memory write acknowledge
signal empty_256: std_logic; -- signal to indicate that the FIFO memory is empty
signal valid_256: std_logic; -- signal to indicate that the FIFO memory data output can be read
signal FIFO_dout_16: std_logic_vector(15 downto 0); -- 272 bits vector that contains a message stored in FIFO
signal full_16: std_logic; -- signal to indicate that the FIFO memory is full
signal wr_ack_16: std_logic; -- FIFO memory write acknowledge
signal empty_16: std_logic; -- signal to indicate that the FIFO memory is empty
signal valid_16: std_logic; -- signal to indicate that the FIFO memory data output can be read
signal full: std_logic; -- signal to indicate that the FIFO memory is full
signal wr_ack: std_logic; -- FIFO memory write acknowledge
signal empty: std_logic; -- signal to indicate that the FIFO memory is empty
signal valid: std_logic; -- signal to indicate that the FIFO memory data output can be read
-- signal to indicate that any message has been lost because the FIFO memories are full and it can't be stored
signal lost_msg: std_logic;

begin
-- INSTANTIATIONS
HEX2ASCII_conv : HEX_TO_ASCII_CONVERTER
   PORT MAP (
    CLK => CLK,
     RST => RESET,
     DIN => r_data2conv,
     ENABLE => enable_conv,
     DOUT => r_data_conv_o,
     DOUT_VALID => conv_valid
);
BIN2DEC_conv : BIN_TO_DEC_CONVERTER
   PORT MAP( CLK => CLK,
             RST => RESET,
               DIN => r_data2tx,
ENABLE => enable_bcd,
             DOUT => r_bcd_o,
             DOUT_VALID => bcd_valid
);

UART_Transmitter : UART
 GENERIC MAP ( BRDIVISOR => 1302) -- 9600 baudios
 PORT MAP (
    BR_Clk_I => CLK,
    reset    => RESET,
    TxD_PAD_O => TX_PAD,
    RxD_PAD_I => '1',
    byteFromPC => open,
    LoadTx    => load_tx,
    Byte2PC   => r_data2tx,
    TxBusy    => tx_busy,
    RxAv      => open
    );

FIFO_16BITS : FIFO_16B
 PORT MAP (
   clk => CLK,
   rst => RESET,
   din => FIFO_16_IN,
   wr_en => wr_en,
   rd_en => rd_en,
   dout => FIFO_dout_16,
   full => full_16,
```

```
   wr_ack => wr_ack_16,
   empty => empty_16,
   valid => valid_16
  );

FIFO_256BITS : FIFO_256B
  PORT MAP (
   clk => CLK,
   rst => RESET,
   din => FIFO_256_IN,
   wr_en => wr_en,
   rd_en => rd_en,
   dout => FIFO_dout_256,
   full => full_256,
   wr_ack => wr_ack_256,
   empty => empty_256,
   valid => valid_256
  );

full <= full_256 and full_16; -- Both FIFO memories are full
wr_ack <= wr_ack_256 and wr_ack_16; -- Both FIFO memories have received the data on its input
empty <= empty_256 and empty_16; -- Both FIFO memories are empty
valid <= valid_256 and valid_16;          -- The data output of FIFO memories could be read

write_FIFOs_process:process (RESET,CLK)
begin
if RESET= '1' then
wr_en <= '0';
            lost_msg <= '0';
elsif CLK'event and clk = '1' then
            if NEW_DATA = '1' and full = '0' then -- If a new message is received and the FIFO memories aren't full
                        wr_en <= '1'; -- The write operation is actived for both FIFO memories
            else
                        wr_en <= '0';
            end if;
then -- If a new message is received and the FIFO memories are full this message has been lost
            if NEW_DATA = '1' and full = '1'
                        lost_msg <= '1';
            end if;
end if;
end process;

r_128b_process:process (RESET,CLK)
begin
if RESET= '1' then
            r_128b <= (others=>'0');
elsif CLK'event and clk = '1' then
            if load_master = '1' then
                        r_128b <= FIFO_dout_256 (255 downto 128);
            elsif load_slave = '1' then
                        r_128b <= FIFO_dout_256 (127 downto 0);
            elsif shift_4 = '1' then
                        r_128b <= r_128b (123 downto 0) & "0000" ;
            elsif shift_1 = '1' then
                        r_128b <= '0' & r_128b (126 downto 0);
            elsif shift_2 = '1' then
                        r_128b <= "00" & r_128b (125 downto 0);
            elsif shift_3 = '1' then
                        r_128b <= "000" & r_128b (124 downto 0);
            else
                        r_128b <= r_128b;
            end if;
```

```
end if;
end process;

r_12b_process:process (RESET,CLK)
begin
if RESET= '1' then
        r_12b <= (others=>'0');
elsif CLK'event and clk = '1' then
        if r2_bcd_valid = '1' then
                r_12b <= r_bcd_o;
        elsif shift_bcd = '1' then
                r_12b <= r_12b (7 downto 0) & "0000" ;
        else
                r_12b <= r_12b;
        end if;
end if;
end process;

aux_cnt_process:process (RESET,CLK)
begin
if RESET= '1' then
        aux_cnt <= (others=>'0');
elsif CLK'event and clk = '1' then
        if load_128 = '1' then
                aux_cnt <= "01111111";
        elsif load_2 = '1' then
                aux_cnt <= "00000010";
        elsif substract_1 = '1' then
                aux_cnt <= aux_cnt - 1;
        elsif substract_4 = '1' then
                aux_cnt <= aux_cnt - 4;
        else
                aux_cnt <= aux_cnt;
        end if;
end if;
end process;

pos_cab_process:process (RESET,CLK)
begin
if RESET= '1' then
pos_cab <= (others=>'0');
elsif CLK'event and clk = '1' then
        if add_pos = '1' then
                pos_cab <= pos_cab + 1;
        elsif rst_pos = '1' then
                pos_cab <= (others=>'0');
        else
                pos_cab <= pos_cab;
        end if;
end if;
end process;

msg_cnt_process:process (RESET,CLK)
begin
if RESET= '1' then
        msg_cnt <= (others=>'0');
elsif CLK'event and clk = '1' then
        if add_msg = '1' then
                msg_cnt <= msg_cnt + 1;
        else
                msg_cnt <= msg_cnt;
        end if;
```

```
end if;
end process;

registers_process: process (RESET,CLK)
begin
if RESET= '1' then
		r_data2conv <= (others=>'0');
		r_data2tx <= (others=>'0');
		r_control <= (others=>'0');
		r_bcd_valid <= '0';
		r2_bcd_valid <= '0';
		r_tx_end <= '0';
elsif CLK'event and clk = '1' then
		r_data2conv <= nr_data2conv;
		r_data2tx <= nr_data2tx;
		r_control <= nr_control;
		r_bcd_valid <= bcd_valid;
		r2_bcd_valid <= r_bcd_valid;
		r_tx_end <= nr_tx_end;
end if;
end process;

sinc_FSM_process: process (RESET,CLK)
begin
if RESET= '1' then
		r_state <= idle;
elsif CLK'event and clk = '1' then
		r_state <= nr_state;
end if;
end process;
fsm_process: process (r_state, empty, pos_cab, r2_bcd_valid, conv_valid, aux_cnt, FIFO_dout_16, r_control, valid,
tx_busy,msg_cnt, r_tx_end, r_data2tx, r_data2conv, r_data_conv_o, r_12b, r_128b)
begin
nr_state <= r_state;
rd_en <= '0';
load_tx <= '0';
enable_bcd <= '0';
enable_conv <= '0';
nr_data2tx <= r_data2tx;
nr_data2conv <= r_data2conv;
nr_control <= r_control;
shift_bcd <= '0';
load_master <= '0';
load_slave <= '0';
shift_4 <= '0';
shift_1 <= '0';
shift_2 <= '0';
shift_3 <= '0';
substract_4 <= '0';
substract_1 <= '0';
load_128 <= '0';
load_2 <= '0';
shift_4 <= '0';
add_pos <= '0';
rst_pos <= '0';
add_msg <= '0';
nr_tx_end <= r_tx_end;
		case r_state is

		when idle =>
			if empty = '0' then -- any message to send is contained in the FIFO memory
				nr_state <= send_cab;
```

```
		end if;
	when send_cab =>
		nr_state <= uart_and_control;
		load_tx <= '1';
		case pos_cab is
			when "0000" =>
				nr_data2tx <= conv_std_logic_vector(77,8); -- M
				nr_control <= "0000"; -- next state is send_cab
			when "0001" =>
				nr_data2tx <= conv_std_logic_vector(83,8); -- S
				nr_control <= "0000"; -- next state is send_cab
			when "0010" =>
				nr_data2tx <= conv_std_logic_vector(71,8); -- G
				nr_control <= "0001"; -- next state is send_bcd (msg_number)
			when "0011" =>
				nr_data2tx <= conv_std_logic_vector(58,8); -- :
				nr_control <= "0000"; -- next state is send_cab
			when "0100" =>
				nr_data2tx <= conv_std_logic_vector(77,8); -- M
				nr_control <= "0000"; -- next state is send_cab
			when "0101" =>
				nr_data2tx <= conv_std_logic_vector(45,8); -- -
				nr_control <= "0100"; -- next state is data_master
			when "0110" =>
				nr_data2tx <= conv_std_logic_vector(45,8); -- -
				nr_control <= "0000"; -- next state is send_cab
			when "0111" =>
				nr_data2tx <= conv_std_logic_vector(83,8); -- S
				nr_control <= "0000"; -- next state is send_cab
			when "1000" =>
				nr_data2tx <= conv_std_logic_vector(45,8); -- -
				nr_control <= "0111"; -- next state is data_slave
			when "1001" =>
				nr_data2tx <= conv_std_logic_vector(45,8); -- -
				nr_control <= "0000"; -- next state is send_cab
			when "1010" =>
				nr_data2tx <= conv_std_logic_vector(66,8); -- B
				nr_control <= "1010"; -- next state is send_bcd (number of bits)
			when "1011" =>
				nr_data2tx <= conv_std_logic_vector(45,8); -- -
				nr_control <= "0000"; -- next state is send_cab
			when "1100" =>
				nr_data2tx <= conv_std_logic_vector(80,8); -- P
				nr_control <= "1011"; -- next state is send_pol_and_phase
			when "1101" =>
				nr_data2tx <= conv_std_logic_vector(45,8); -- -
				nr_control <= "0000"; -- next state is send_cab
			when "1110" =>
				nr_data2tx <= conv_std_logic_vector(80,8); -- P
				nr_control <= "0000"; -- next state is send_cab
			when others =>
				nr_data2tx <= conv_std_logic_vector(72,8); -- H
				nr_control <= "1101"; -- next state is idle
		end case;
	when conv_ASCII =>
		if conv_valid = '1' then
			nr_state <= uart_and_control;
			load_tx <= '1';
			nr_data2tx <= r_data_conv_o;
		else
			nr_state <= conv_ASCII;
			load_tx <= '0';
```

```
                        nr_data2tx <= r_data2tx;
                end if;
        when send_bcd =>
                if aux_cnt = 0 then
                        nr_control <= "0011"; -- next state is send_cab
                        nr_state <= conv_ASCII;
                        nr_data2conv <= r_12b (11 downto 8);
                        enable_conv <= '1';
                        substract_1 <= '0';
                else -- aux_cnt > 0
                        nr_control <= "0010"; -- next state is send_bcd
                        nr_state <= conv_ASCII;
                        nr_data2conv <= r_12b (11 downto 8);
                        enable_conv <= '1';
                        substract_1 <= '1';
                end if;
        when shift_128b =>
-- the register is shifted until the first data is situated on the high possition
                if ((aux_cnt + 1) - unsigned(FIFO_dout_16(15 downto 8))) >= 4 then
                        substract_4 <= '1';
                        shift_4 <= '1';
                        shift_1 <= '0';
                        shift_2 <= '0';
                        shift_3 <= '0';
                        nr_state <= shift_128b;
                elsif ((aux_cnt + 1) - unsigned(FIFO_dout_16(15 downto 8))) = 3 then
                        substract_4 <= '0';
                        shift_4 <= '0';
                        shift_1 <= '0';
                        shift_2 <= '0';
                        shift_3 <= '1';
                        nr_state <= message_Data;
                elsif ((aux_cnt + 1) - unsigned(FIFO_dout_16(15 downto 8))) = 2 then
                        substract_4 <= '0';
                        shift_4 <= '0';
                        shift_1 <= '0';
                        shift_2 <= '1';
                        shift_3 <= '0';
                        nr_state <= message_Data;
                elsif ((aux_cnt + 1) - unsigned(FIFO_dout_16(15 downto 8))) = 1 then
                        substract_4 <= '0';
                        shift_4 <= '0';
                        shift_1 <= '1';
                        shift_2 <= '0';
                        shift_3 <= '0';
                        nr_state <= message_Data;
                else
                        substract_4 <= '0';
                        substract_1 <= '0';
                        shift_4 <= '0';
                        shift_1 <= '0';
                        shift_2 <= '0';
                        shift_3 <= '0';
                        nr_state <= message_Data;
                end if;

        when message_Data =>
                if aux_cnt > 4 then
                        substract_4 <= '1';
                        substract_1 <= '0';
                        shift_4 <= '1';
                        shift_1 <= '0';
```

```
                        nr_state <= conv_ASCII;
                        enable_conv <= '1';
                        nr_data2conv <= r_128b (127 downto 124);
                        nr_control <= "0101";
                else
                        substract_4 <= '0';
                        substract_1 <= '0';
                        shift_4 <= '0';
                        shift_1 <= '0';
                        nr_state <= conv_ASCII;
                        enable_conv <= '1';
                        nr_data2conv <= r_128b (127 downto 124);
                        nr_control <= "0110";
                end if;
        when send_pol_and_phase =>
                if r_control = "1011" then -- polarity must be sent
                        nr_state <= conv_ASCII;
                        enable_conv <= '1';
                        nr_data2conv <= FIFO_dout_16(7 downto 4);
                        nr_control <= "1100";
                else -- phase must be sent
                        nr_state <= conv_ASCII;
                        enable_conv <= '1';
                        nr_data2conv <= FIFO_dout_16(3 downto 0);
                        nr_control <= "1110";
                end if;
        when others => -- uart_and_control
                if tx_busy = '0' then
                        case r_control is
                                when "0000" =>
                                        add_pos <= '1';
                                        rst_pos <= '0';
                                        nr_state <= send_cab;
                                        enable_bcd <= '0';
                                when "0001" =>
                                        add_pos <= '0';
                                        rst_pos <= '0';
                                        enable_bcd <= '1';
                                        nr_data2tx <= conv_std_logic_vector(msg_cnt,8);
                                        load_2 <= '1';
                                        if r2_bcd_valid = '1' then -- wait until the conversion has finished
                                                nr_state <= send_bcd;
                                        else
                                                nr_state <= uart_and_control;
                                        end if;
                                when "0010" =>
                                        shift_bcd <= '1';
                                        add_pos <= '0';
                                        rst_pos <= '0';
                                        nr_state <= send_bcd;
                                        enable_bcd <= '0';
                                when "0011" =>
                                        nr_state <= send_cab;
                                        add_pos <= '1';
                                when "0100" =>
                                        if valid = '1' then -- FIFOS outputs could be read
                                                rd_en <= '0';
                                                nr_state <= shift_128b;
                                                load_master <= '1';
                                                load_128 <= '1';
                                        else    -- wait until the data could be read
                                                rd_en <= '1';
```

```
                                        nr_state <= uart_and_control;
                                        load_128 <= '0';
                                end if;
                        when "0101" =>
                                nr_state <= message_Data;
                        when "0110" =>
                                add_pos <= '1';
                                nr_state <= send_cab;
                        when "0111" =>
                                nr_state <= shift_128b;
                                load_slave <= '1';
                                load_128 <= '1';
                        when "1000" =>
                                nr_state <= message_Data;
                        when "1001" =>
                                nr_state <= send_cab;
                        when "1010" =>
                                add_pos <= '0';
                                rst_pos <= '0';
                                enable_bcd <= '1';
                                nr_data2tx <= FIFO_dout_16(15 downto 8);
                                load_2 <= '1';
                                if r2_bcd_valid = '1' then -- wait until the conversion has finished
                                        nr_state <= send_bcd;
                                else
                                        nr_state <= uart_and_control;
                                end if;
                        when "1011" =>
                                nr_state <= send_pol_and_phase;
                        when "1100" =>
                                nr_state <= send_cab;
                                add_pos <= '1';
                        when "1101" =>
                                nr_state <= send_pol_and_phase;
                        when "1110" =>
                                nr_state <= uart_and_control;
                                load_tx <= '1';
                                if r_tx_end = '1' then
                                        nr_data2tx <= conv_std_logic_vector(13,8); -- CR
                                        nr_control <= "1111";
                                        nr_tx_end <= '0';
                                else
                                        nr_data2tx <= conv_std_logic_vector(10,8); -- End of line
                                        nr_control <= "1110";
                                        nr_tx_end <= '1';
                                end if;
                        when others =>
                                nr_state <= idle;
                                add_msg <= '1';
                                rst_pos <= '1';
                        end case;
                else
                        nr_state <= uart_and_control;
                end if;
 end case;
end process;

--OUTPUTS
LED_EMPTY <= empty;
LED_MSG_LOST <= lost_msg;
end Behavioral;
```

11.4.5 Top I2C

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity TOP_I2C is
   Port ( CLK : in  STD_LOGIC;
          RESET : in  STD_LOGIC;
          SDA_in : in  STD_LOGIC;
          SCL_in : in  STD_LOGIC;
          LED_EMPTY : out  STD_LOGIC;
          LED_MSG_LOST : out  STD_LOGIC;
          LED_RUNNING : out  STD_LOGIC;
          TX_pad : out  STD_LOGIC);
end TOP_I2C;

architecture Behavioral of TOP_I2C is
-- COMPONENTS
COMPONENT CONTROL_I2C -- block to control the transmission of each captured message by mean of an UART
   Port ( CLK : in  STD_LOGIC;
          RESET : in  STD_LOGIC;
          NEW_BYTE : in  STD_LOGIC;
          NEW_EOM : in  STD_LOGIC;
          TX_PAD : out  STD_LOGIC;
          LED_EMPTY : out  STD_LOGIC;
          LED_MSG_LOST : out  STD_LOGIC;
          FIFO_8_IN : in  STD_LOGIC_VECTOR (7 downto 0);
          FIFO_2_IN : in  STD_LOGIC_VECTOR (1 downto 0));
end COMPONENT;

COMPONENT I2C_sampler -- Component whose function is to capture messages sent by the I2C protocol.
   Port ( CLK : in  STD_LOGIC;
          RESET : in  STD_LOGIC;
          SDA : in  STD_LOGIC;
          SCL : in  STD_LOGIC;
          BYTE_RECEIVED : out  STD_LOGIC_VECTOR(7 downto 0);
          NEW_BYTE : out  STD_LOGIC;
          EOM : out  STD_LOGIC;
          ACK : out  STD_LOGIC;
          NEW_EOM : out  STD_LOGIC;
          RUNNING : out  STD_LOGIC);
end COMPONENT;

-- Signals
signal n_byte: std_logic;
signal n_eom: std_logic;
signal byte_rec: std_logic_vector (7 downto 0);
signal s_eom: std_logic;
signal s_ack: std_logic;
signal ack_eom: std_logic_vector (1 downto 0);

begin
-- Combinational logic
ack_eom <= s_ack & s_eom;
-- INSTANTIATIONS
Control_block: CONTROL_I2C -- block to control the transmission of each captured message by mean of an UART
   Port map ( CLK => CLK,
          RESET => RESET,
          NEW_BYTE => n_byte,
          NEW_EOM => n_eom,
          TX_PAD => TX_pad,
          LED_EMPTY => LED_EMPTY,
```

```
        LED_MSG_LOST => LED_MSG_LOST,
        FIFO_8_IN => byte_rec,
        FIFO_2_IN => ack_eom
);

Sampler: I2C_sampler -- Component whose function is to capture messages sent by the I2C protocol.
   Port map ( CLK => CLK,
        RESET => RESET,
        SDA => SDA_in,
        SCL => SCL_in,
        BYTE_RECEIVED => byte_rec,
        NEW_BYTE => n_byte,
        EOM => s_eom,
        ACK => s_ack,
        NEW_EOM => n_eom,
        RUNNING => LED_RUNNING
);

end Behavioral;
```

11.4.6 Bloque I2C_Sampler

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

entity I2C_sampler is
   Port ( CLK : in  STD_LOGIC;
        RESET : in  STD_LOGIC;
        SDA : in  STD_LOGIC;
        SCL : in  STD_LOGIC;
        BYTE_RECEIVED : out  STD_LOGIC_VECTOR(7 downto 0);
        NEW_BYTE : out  STD_LOGIC;
        EOM : out  STD_LOGIC;
        ACK : out  STD_LOGIC;
        NEW_EOM : out  STD_LOGIC;
        RUNNING : out  STD_LOGIC);
end I2C_sampler;
architecture Behavioral of I2C_sampler is
--types
--FSM
type state is (idle, read_byte, read_ack);
 signal r_state : state;
 signal nr_state : state;
-- Counters
-- Counter to calculate the number of bits received during a byte transmission
signal bits_cnt: unsigned (3 downto 0);
signal add_bit: std_logic; -- Signal to increment the counter
signal reset_cnt: std_logic; -- Signal to set the counter to 0
signal Byte_complete: std_logic; -- Signal to identify that 8 bits have been received
-- Control
signal r_first_byte: std_logic; -- Signal to identify if a Byte is the first transmitted during a transmission
signal nr_first_byte: std_logic;
signal start: std_logic; -- Signal to identify a start condition
signal stop: std_logic; -- Signal to identify a stop condition
signal r_byte: std_logic_vector (7 downto 0);-- 8 bits register to store the byte received
signal reset_byte: std_logic; -- signal to set all bits to '0' to get a new byte
signal load_bit: std_logic; -- signal to load a new bit in the 8 bits register
signal r_ack: std_logic; -- Signal to store the Acknowledge value
signal nr_ack: std_logic;
signal r_eom: std_logic; -- Signal to identify if a Byte received is the end of a message.
signal nr_eom: std_logic;
```

```
signal r_new_eom: std_logic; -- Signal to identify a new end of message value
signal nr_new_eom: std_logic;
signal s_new_byte: std_logic;
signal stop_error: std_logic; -- Signal to indicate that the transmission has been stopped during a byte reception
signal s_running: std_logic;                    -- Signal to identify that the module is analyzing a message
-- SCL
signal r_scl: std_logic;-- Signal to store the SCL line value
signal r2_scl: std_logic;-- Signal to store the SCL line value
signal rising_scl: std_logic;-- Signal to identify a rising edge on the SCL line
-- SDA
signal r_sda: std_logic;-- Signal to store the SDA line value
signal r2_sda: std_logic;-- Signal to store the SDA line value
signal rising_sda: std_logic;-- Signal to identify a rising edge on the SDA line
signal falling_sda: std_logic;-- Signal to identify a falling edge on the SDA line

begin
-- Combinational logic
rising_scl <= '1' when (r2_scl = '0' and r_scl = '1') else '0';
rising_sda <= '1' when (r2_sda = '0' and r_sda = '1') else '0';
falling_sda <= '1' when (r2_sda = '1' and r_sda = '0') else '0';
start <= '1' when (r2_scl = '1' and falling_sda = '1') else '0';
stop <= '1' when (r2_scl = '1' and rising_sda = '1') else '0';

register_process:
 process(RESET,CLK)
 begin
 if RESET = '1' then
  r_scl <= '1'; -- Idle state of SCL line
                r2_scl <= '1';
                r_sda <= '1'; -- Idle state of SDA line
                r2_sda <= '1';
                r_first_byte <= '1';
                r_ack <= '0';
                r_eom <= '0';
                r_new_eom <= '0';
                BYTE_RECEIVED <= (others=>'0');
    elsif CLK'event and CLK='1' then
                 r_scl <= SCL; -- SCL input register
                r2_scl <= r_scl;
                r_sda <= SDA; -- SDA input register
                r2_sda <= r_sda;
                r_first_byte <= nr_first_byte;
                r_ack <= nr_ack;
                r_eom <= nr_eom;
                r_new_eom <= nr_new_eom;
                if s_new_byte = '1' then
                        BYTE_RECEIVED <= r_byte; -- Captured byte is sent to the control unit
               end if;
end if;
end process;

Byte_register_process:
process(RESET,CLK)
 begin
  if RESET = '1' then
              r_byte <= (others=>'0');
  elsif CLK'event and CLK='1' then
             if load_bit = '1' then
                        r_byte <= r_byte (6 downto 0) & r2_sda;
                        elsif reset_byte = '1' then
                                r_byte <= (others=>'0');
               else
```

```
                        r_byte <= r_byte;
                end if;
 end if;
 end process;

Bits_counter_process:
process(RESET,CLK)
begin
if RESET = '1' then
                bits_cnt <= (others=>'0');
                Byte_complete <= '0';
elsif CLK'event and CLK='1' then
 if add_bit = '1' then
                        bits_cnt <= bits_cnt + 1;
                elsif reset_cnt = '1' then
                        bits_cnt <= (others=>'0');
                else
                        bits_cnt <= bits_cnt;
                end if;
                if bits_cnt = 8 then
                        Byte_complete <= '1';
                else
                        Byte_complete <= '0';
                end if;
end if;
end process;

fsm_process: -- FSM state register
process(RESET,CLK)
begin
if RESET = '1' then
                r_state <= idle;
 elsif CLK'event and CLK='1' then
                r_state <= nr_state;
 end if;
 end process;

fsm_transitions:  -- FSM states transitions and outputs
process ( r_state, start, rising_scl, Byte_complete, r_first_byte, bits_cnt, r2_sda, stop_error)
begin
 -- Default values
 nr_state   <= r_state;
add_bit <= '0';
reset_cnt <= '0';
load_bit <= '0';
nr_first_byte <= r_first_byte;
nr_ack <= r_ack;
nr_eom <= r_eom;
s_new_byte <= '0';
nr_new_eom <= '0';
s_running <= '0';
stop_error <= '0';
reset_byte <= '0';
                case r_state is
when idle =>
                        s_running <= '0'; -- The analyzer is stopped
                        if start = '1' then -- Wait until a start condition is received
                                nr_state <= read_byte; -- A new Byte will be read
                        else
                                nr_state <= idle;
                        end if;
                when read_byte =>
```

```
                s_running <= '1'; -- the analyzer is receiving data
                if rising_scl = '1' then -- When the SCL line is on high
                        load_bit <= '1'; -- a New bit is read
                        add_bit <= '1'; -- the bits counter is incremented
                elsif Byte_complete = '1' then -- a complete byte has been received
                        nr_state <= read_ack;
                        nr_first_byte <= '0'; -- the First Byte of a transmission has been read
                        s_new_byte <= '1'; -- the Byte received can be read
                        reset_cnt <= '1';
                elsif stop_error = '1' then -- An error has been detected during the transmission
                        nr_state <= idle; -- The system returns to the idle state
                        s_new_byte <= '1'; -- the Byte received can be read
                        reset_cnt <= '1';
                else
                        load_bit <= '0';
                        add_bit <= '0';
                        nr_state <= read_byte;
                end if;
                if start = '1' then -- If the first byte has been received and
                        if bits_cnt = 1 then -- restart is received after a complete byte transmission
                                nr_first_byte <= '1'; -- The First Byte of a new transmission should be read
                                reset_cnt <= '1'; -- the bits counter is set to 0 to read a new Byte
                                nr_new_eom <= '1';
                                nr_eom <= '1'; -- The last byte received was the end of a message
                        else -- the transmission has been interrupted
                                nr_first_byte <= '1'; -- The First Byte of a new transmission should be read
                                reset_cnt <= '1'; -- the bits counter is set to 0 to read a new Byte
                                nr_new_eom <= '1';
                                nr_eom <= '1'; -- The last byte received was the end of a message
                                nr_ack <= '1'; -- the receiver can´t send an Ack if the transmission is stopped
                                stop_error <= '1'; -- error is detected
                                reset_byte <= '1'; -- the register is prepared to receive a new byte
                        end if;
                elsif stop = '1' then -- If a stop condition is identified it means that it was the end of a message
                        if bits_cnt = 1 then -- stop is received after a complete byte transmission
                                nr_state <= idle;
                                nr_first_byte <= '1'; -- The First Byte of a new transmission should be read
                                reset_cnt <= '1'; -- the bits counter is set to 0 to read a new Byte
                                nr_new_eom <= '1';
                                nr_eom <= '1'; -- The last byte received was the end of a message
                        else
                                nr_state <= idle;
                                nr_first_byte <= '1'; -- The First Byte of a new transmission should be read
                                reset_cnt <= '1'; -- the bits counter is set to 0 to read a new Byte
                                nr_new_eom <= '1';
                                nr_eom <= '1'; -- The last byte received was the end of a message
                                stop_error <= '1'; -- error is detected
                                reset_byte <= '1'; -- the register is prepared to receive a new byte
                                nr_ack <= '1'; -- the receiver can´t send an Ack if the transmission is stopped
                        end if;
-- When the counter is higher than 0 it means that a new Byte is been received and the last one stored wasn´t the
end of a message
        elsif r_first_byte = '0' and bits_cnt = 1 then
                        nr_new_eom <= '1';
                        nr_eom <= '0'; -- The last byte received wasn´t the end of a message
        end if;
        when read_ack =>
                s_running <= '1'; -- the analyzer is receiving data
                if rising_scl = '1' then -- When the SCL line is on high
                        nr_ack <= r2_sda; -- the Acknowledge value is stored
                        nr_state <= read_byte; -- A new Byte will be read
                        reset_byte <= '1'; -- the register is prepared to receive a new byte
```

```
                else
                        nr_state <= read_ack;
                        reset_byte <= '0';
                end if;
        when others =>
                nr_state <= idle;
        end case;
end process;

-- OUTPUTS ASSIGNMENT
        NEW_EOM <= '1' when r_new_eom = '1' and nr_new_eom = '0' else '0';
        NEW_BYTE <= s_new_byte;
        ACK <= r_ack;
        EOM <= r_eom;
        RUNNING <= s_running;
end Behavioral;
```

11.4.7 Bloque de control I2C

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
entity CONTROL_I2C is
   Port ( CLK : in  STD_LOGIC;              -- Main clock port
          RESET : in  STD_LOGIC;            -- Reset actived high port
          NEW_BYTE : in  STD_LOGIC;         -- Input from the sampler which indicates that a byte has been received
-- Input from the sampler which indicates that an ack and the end information of a message has been received
          NEW_EOM : in  STD_LOGIC;
          TX_PAD : out  STD_LOGIC;          -- RS-232 serial output to comunicate with the PC
-- Output connected to a LED to recognize that the FIFO memories are empty. It means that there are no messages
to send
          LED_EMPTY : out  STD_LOGIC;
-- Output connected to a LED to indicate that any message has been lost because when it was received the FIFO
memories were full.
          LED_MSG_LOST : out  STD_LOGIC;
          FIFO_8_IN : in  STD_LOGIC_VECTOR (7 downto 0);    -- FIFO 8 bits data input (byte of a message)
-- FIFO 2 bits data input (acknowledge and end of message information)
          FIFO_2_IN : in  STD_LOGIC_VECTOR (1 downto 0));
end CONTROL_I2C;

architecture Behavioral of CONTROL_I2C is
-- COMPONENTS
COMPONENT FIFO_8B -- 8 bits FIFO memory with depth = 1024 messages
  PORT (
    clk : IN STD_LOGIC;
    rst : IN STD_LOGIC;
    din : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
    wr_en : IN STD_LOGIC;
    rd_en : IN STD_LOGIC;
    dout : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
    full : OUT STD_LOGIC;
    wr_ack : OUT STD_LOGIC;
    empty : OUT STD_LOGIC;
    valid : OUT STD_LOGIC
  );

END COMPONENT;
COMPONENT FIFO_2B -- 2 bits FIFO memory with depth = 1024 messages
  PORT (
    clk : IN STD_LOGIC;
    rst : IN STD_LOGIC;
    din : IN STD_LOGIC_VECTOR(1 DOWNTO 0);
```

```
    wr_en : IN STD_LOGIC;
    rd_en : IN STD_LOGIC;
    dout : OUT STD_LOGIC_VECTOR(1 DOWNTO 0);
    full : OUT STD_LOGIC;
    wr_ack : OUT STD_LOGIC;
    empty : OUT STD_LOGIC;
    valid : OUT STD_LOGIC
  );

END COMPONENT;

-- This component converts a 4 bits vector(hexadecimal bit) to its correspondent ASCII value
COMPONENT HEX_TO_ASCII_CONVERTER
    PORT ( CLK : in  STD_LOGIC;
           RST : in  STD_LOGIC;
           DIN : in  STD_LOGIC_VECTOR (3 downto 0);
           ENABLE : in  STD_LOGIC;
           DOUT : out  STD_LOGIC_VECTOR (7 downto 0);
           DOUT_VALID : out  STD_LOGIC);
end COMPONENT;

COMPONENT BIN_TO_DEC_CONVERTER -- This component receives a binary data and returns its value in bcd
    PORT ( CLK : in  STD_LOGIC;
           RST : in  STD_LOGIC;
           DIN : in  STD_LOGIC_VECTOR (7 downto 0);
           ENABLE : in  STD_LOGIC;
           DOUT : out  STD_LOGIC_VECTOR (11 downto 0);
           DOUT_VALID : out  STD_LOGIC);
end COMPONENT;

COMPONENT UART -- RS_232 transmitter
  generic(BRDIVISOR: INTEGER range 0 to 65535);-- := 1302);--130); -- Baud rate divisor
  port (
     BR_Clk_I : in  std_logic;  -- Clock used for Transmit/Receive
     reset    : in  std_logic;
     TxD_PAD_O: out std_logic;  -- Tx RS232 Line
     RxD_PAD_I: in  std_logic;  -- Rx RS232 Line
     byteFromPC: out std_logic_vector(7 downto 0);
     LoadTx    : in  std_logic;
     Byte2PC   : in std_logic_vector(7 downto 0);
     TxBusy    : out std_logic;  -- Transmiter Busy
     RxAv      : out std_logic  -- Data Received
     );
end COMPONENT;
-- TYPES
type state is (idle, send_cab, send_bcd, conv_ASCII, uart_and_control, send_address, send_byte, send_ack); -- FSM
states
-- SIGNALS
-- FSM signals
signal r_state : state; -- Actual state signal
signal nr_state: state; -- Next state signal
signal r_control : std_logic_vector (3 downto 0); -- signal to control transitions
signal nr_control : std_logic_vector (3 downto 0);
-- Counters
signal pos_cab: unsigned (3 downto 0); -- Counter to send the correct header character
signal add_pos: std_logic; -- signal to increment the position counter
signal rst_pos: std_logic; -- signal to reset the position counter
signal bit_pos: std_logic; -- signal to set the position counter for a new byte transmission
-- Auxiliar counter used for sending the message count value with hundreds, tens and units
signal aux_cnt: unsigned (1 downto 0);
signal load_2: std_logic; -- signal to load the value to the counter
signal substract_1: std_logic; -- signal to decrement 1 the counter value
```

```
-- signal to control if the 4 bits group that will be sent is the beginning or the end of a byte
signal r_complete_byte: std_logic;
signal nr_complete_byte: std_logic;
signal msg_cnt: unsigned (7 downto 0); -- Counter of messages sent (limit = 256)
signal add_msg: std_logic; -- signal to increment the position counter
-- Registers
-- 8 bits
signal r_8b: std_logic_vector (7 downto 0); -- register used for load the Byte read from the FIFO memory
signal load_byte: std_logic; -- signal to load the master data
signal shift_4: std_logic; -- signal to shift the register 4 positions
-- 12 bits
signal r_12b: std_logic_vector (11 downto 0); -- register used for load bcd results
signal shift_bcd: std_logic; -- signal to shift the register 4 positions
-- BCD conversor
signal r_bcd_o: std_logic_vector (11 downto 0); -- register used for load bcd results
signal enable_bcd: std_logic; -- signal to enable the binary to decimal converter
signal bcd_valid: std_logic; -- signal to load the binary to decimal result
signal r_bcd_valid: std_logic; -- signal to load the binary to decimal result
signal r2_bcd_valid: std_logic; -- signal to load the binary to decimal result
-- ASCII conversor
signal r_data2conv: std_logic_vector (3 downto 0); -- register used for load a 4 bits gropu to convert
signal nr_data2conv: std_logic_vector (3 downto 0); -- register used for load a 4 bits gropu to convert
signal r_data_conv_o: std_logic_vector(7 downto 0); -- 8 bits vector that contains data converted
signal enable_conv: std_logic; -- signal to enable the hex to ASCII converter
signal conv_valid: std_logic; -- signal to load the hex to ASCII result
-- UART
signal r_data2tx: std_logic_vector(7 downto 0); -- 8 bits vector that contains data to send
signal nr_data2tx: std_logic_vector(7 downto 0);
signal load_tx: std_logic; -- This signal indicates that a new transmission is required
signal tx_busy: std_logic; -- Signal which indicates that the UART is busy
signal r_tx_end: std_logic; -- Signal which indicates that the message is sent and a carriage return must be sent
signal nr_tx_end: std_logic;
--FIFOs
signal wr_en_8: std_logic; -- Write enable signal of FIFO_8B memory
signal wr_en_2: std_logic; -- Write enable signal of FIFO_2B memory
signal rd_en: std_logic; -- Read enable signal of FIFO memory
-- 2 bits vector that contains the acknownledge and the EOM information of a byte stored in FIFO_8B Memory
signal FIFO_dout_2: std_logic_vector(1 downto 0);
signal full_2: std_logic; -- signal to indicate that the FIFO memory is full
signal wr_ack_2: std_logic; -- FIFO memory write acknowledge
signal empty_2: std_logic; -- signal to indicate that the FIFO memory is empty
signal valid_2: std_logic; -- signal to indicate that the FIFO memory data output can be read
-- 8 bits vector that contains a byte from the message stored in FIFO Memory
signal FIFO_dout_8: std_logic_vector(7 downto 0);
signal full_8: std_logic; -- signal to indicate that the FIFO memory is full
signal wr_ack_8: std_logic; -- FIFO memory write acknowledge
signal empty_8: std_logic; -- signal to indicate that the FIFO memory is empty
signal valid_8: std_logic; -- signal to indicate that the FIFO memory data output can be read
signal full: std_logic; -- signal to indicate that the FIFO memory is full
signal wr_ack: std_logic; -- FIFO memory write acknowledge
signal empty: std_logic; -- signal to indicate that the FIFO memory is empty
signal valid: std_logic; -- signal to indicate that the FIFO memory data output can be read
-- signal to indicate that any message has been lost because the FIFO memories are full and it can't be stored
signal lost_msg: std_logic;

begin
-- INSTANTIATIONS
HEX2ASCII_conv : HEX_TO_ASCII_CONVERTER
   PORT MAP (
     CLK => CLK,
     RST => RESET,
     DIN => r_data2conv,
```

```
    ENABLE => enable_conv,
    DOUT => r_data_conv_o,
    DOUT_VALID => conv_valid
);

BIN2DEC_conv : BIN_TO_DEC_CONVERTER
  PORT MAP( CLK => CLK,
       RST => RESET,
       DIN => r_data2tx,
       ENABLE => enable_bcd,
       DOUT => r_bcd_o,
       DOUT_VALID => bcd_valid
);

UART_Transmitter : UART
GENERIC MAP ( BRDIVISOR => 1302)
PORT MAP (
    BR_Clk_I => CLK,
    reset    => RESET,
    TxD_PAD_O => TX_PAD,
    RxD_PAD_I => '1',
    byteFromPC => open,
    LoadTx    => load_tx,
    Byte2PC   => r_data2tx,
    TxBusy    => tx_busy,
    RxAv      => open
    );

FIFO_8BITS : FIFO_8B
 PORT MAP (
   clk => CLK,
   rst => RESET,
   din => FIFO_8_IN,
   wr_en => wr_en_8,
   rd_en => rd_en,
   dout => FIFO_dout_8,
   full => full_8,
   wr_ack => wr_ack_8,
   empty => empty_8,
   valid => valid_8
 );

FIFO_2BITS : FIFO_2B
 PORT MAP (
   clk => CLK,
   rst => RESET,
   din => FIFO_2_IN,
   wr_en => wr_en_2,
   rd_en => rd_en,
   dout => FIFO_dout_2,
   full => full_2,
   wr_ack => wr_ack_2,
   empty => empty_2,
   valid => valid_2
 );

full <= full_2 and full_8;
wr_ack <= wr_ack_2 and wr_ack_8;
empty <= empty_2 and empty_8;
valid <= valid_2 and valid_8;

write_FIFOs_process:process (RESET,CLK)
```

```
begin
if RESET= '1' then
		wr_en_8 <= '0';
		wr_en_2 <= '0';
		lost_msg <= '0';
elsif CLK'event and clk = '1' then
		if NEW_BYTE = '1' and full_8 = '0' then -- A message byte is written in the FIFO_8B memory
			wr_en_8 <= '1';
		else
			wr_en_8 <= '0';
		end if;
-- the ack and the end of message information of a byte is written in the FIFO_2B memory
		if NEW_EOM = '1' and full_2 = '0' then
			wr_en_2 <= '1';
		else
			wr_en_2 <= '0';
		end if;
-- If a new message is received and the FIFO memories are full this message has been lost
		if (NEW_BYTE = '1' and full_8 = '1') or (NEW_EOM = '1' and full_2 = '1') then
			lost_msg <= '1';
		end if;
end if;
end process;

r_8b_process:process (RESET,CLK)
begin
if RESET= '1' then
		r_8b <= (others=>'0');
elsif CLK'event and clk = '1' then
		if load_byte = '1' then
			r_8b <= FIFO_dout_8;
		elsif shift_4 = '1' then
			r_8b <= r_8b (3 downto 0) & "0000" ;
		else
			r_8b <= r_8b;
		end if;
end if;
end process;

r_12b_process:process (RESET,CLK)
begin
if RESET= '1' then
		r_12b <= (others=>'0');
elsif CLK'event and clk = '1' then
		if r2_bcd_valid = '1' then
			r_12b <= r_bcd_o;
		elsif shift_bcd = '1' then
			r_12b <= r_12b (7 downto 0) & "0000" ;
		else
			r_12b <= r_12b;
		end if;
end if;
end process;

aux_cnt_process:process (RESET,CLK)
begin
if RESET= '1' then
		aux_cnt <= (others=>'0');
elsif CLK'event and clk = '1' then
		if load_2 = '1' then
			aux_cnt <= "10";
		elsif substract_1 = '1' then
```

```
                aux_cnt <= aux_cnt - 1;
        else
                aux_cnt <= aux_cnt;
        end if;
end if;
end process;

pos_cab_process:process (RESET,CLK)
begin
if RESET= '1' then
        pos_cab <= (others=>'0');
elsif CLK'event and clk = '1' then
        if add_pos = '1' then
                pos_cab <= pos_cab + 1;
        elsif bit_pos = '1' then -- the counter returns to the init of the transmission of a Byte
                pos_cab <= "1011";
        elsif rst_pos = '1' then
                pos_cab <= (others=>'0');
        else
                pos_cab <= pos_cab;
        end if;
end if;
end process;

msg_cnt_process:process (RESET,CLK)
begin
if RESET= '1' then
        msg_cnt <= (others=>'0');
elsif CLK'event and clk = '1' then
        if add_msg = '1' then
                msg_cnt <= msg_cnt + 1;
        else
                msg_cnt <= msg_cnt;
        end if;
end if;
end process;

registers_process: process (RESET,CLK)
begin
if RESET= '1' then
        r_data2conv <= (others=>'0');
        r_data2tx <= (others=>'0');
        r_control <= (others=>'0');
        r_bcd_valid <= '0';
        r2_bcd_valid <= '0';
        r_complete_byte <= '0';
        r_tx_end <= '0';
elsif CLK'event and clk = '1' then
        r_data2conv <= nr_data2conv;
        r_data2tx <= nr_data2tx;
        r_control <= nr_control;
        r_bcd_valid <= bcd_valid;
        r2_bcd_valid <= r_bcd_valid;
        r_complete_byte <= nr_complete_byte;
        r_tx_end <= nr_tx_end;
end if;
end process;

sinc_FSM_process: process (RESET,CLK)
begin
if RESET= '1' then
        r_state <= idle;
```

```
elsif CLK'event and clk = '1' then
		r_state <= nr_state;
end if;
end process;

fsm_process: process (r_state, empty, pos_cab, r_complete_byte, aux_cnt, FIFO_dout_8, FIFO_dout_2,
r2_bcd_valid, conv_valid, r_control, valid, tx_busy, msg_cnt, r_tx_end)
begin
nr_state <= r_state;
rd_en <= '0';
load_tx <= '0';
enable_bcd <= '0';
enable_conv <= '0';
nr_data2tx <= r_data2tx;
nr_data2conv <= r_data2conv;
nr_control <= r_control;
shift_bcd <= '0';
load_byte <= '0';
shift_4 <= '0';
substract_1 <= '0';
load_2 <= '0';
nr_complete_byte <= r_complete_byte;
add_pos <= '0';
rst_pos <= '0';
add_msg <= '0';
bit_pos <= '0';
nr_tx_end <= r_tx_end;
case r_state is
		when idle =>
			if empty = '0' then -- any message to send is contained in the FIFO memory
				nr_state <= send_cab; -- Firstly the header of a message is sent
			end if;
		when send_cab =>
			nr_state <= uart_and_control;
			load_tx <= '1';
			case pos_cab is
			when "0000" =>
				nr_data2tx <= conv_std_logic_vector(77,8); -- M
					nr_control <= "0000"; -- next state is send_cab
			when "0001" =>
				nr_data2tx <= conv_std_logic_vector(83,8); -- S
				nr_control <= "0000"; -- next state is send_cab
			when "0010" =>
				nr_data2tx <= conv_std_logic_vector(71,8); -- G
				nr_control <= "0001"; -- next state is send_bcd (msg_number)
			when "0011" =>
				nr_data2tx <= conv_std_logic_vector(58,8); -- :
				nr_control <= "0000"; -- next state is send_cab
			when "0100" =>
				nr_data2tx <= conv_std_logic_vector(65,8); -- A
				nr_control <= "0000"; -- next state is send_cab
			when "0101" =>
				nr_data2tx <= conv_std_logic_vector(100,8); -- d
				nr_control <= "0000"; -- next state is send_cab
			when "0110" =>
				nr_data2tx <= conv_std_logic_vector(45,8); -- -
				nr_control <= "0100"; -- next state is send_address
			when "0111" =>
				nr_data2tx <= conv_std_logic_vector(45,8); -- -
				nr_control <= "0000"; -- next state is send_cab
			when "1000" =>
				if FIFO_dout_8(0) = '1' then -- Read operation
```

```
                                        nr_data2tx <= conv_std_logic_vector(82,8); -- R
                                else -- Write operation
                                        nr_data2tx <= conv_std_logic_vector(87,8); -- W
                                end if;
                                nr_control <= "0000"; -- next state is send_cab
                        when "1001" =>
                                nr_data2tx <= conv_std_logic_vector(45,8); -- -
                                nr_control <= "0000"; -- next state is send_cab
                        when "1010" =>
                                nr_data2tx <= conv_std_logic_vector(97,8); -- a
                                nr_control <= "0111"; -- next state is send_ack (address)
                        when "1011" =>
                                nr_data2tx <= conv_std_logic_vector(45,8); -- -
                                nr_control <= "1100"; -- next state is send_byte
                        when "1100" =>
                                nr_data2tx <= conv_std_logic_vector(45,8); -- -
                                nr_control <= "0000"; -- next state is send_cab
                        when "1101" =>
                                nr_data2tx <= conv_std_logic_vector(97,8); -- a
                                nr_control <= "0111"; -- next state is send_ack (byte)
--                      when "1110" =>
--                              nr_data2tx <= conv_std_logic_vector(80,8); -- P
--                              nr_control <= "0000"; -- next state is send_cab
                        when others =>
                                nr_data2tx <= conv_std_logic_vector(72,8); -- H
                                nr_control <= "1101"; -- next state is idle
                        end case;
                when conv_ASCII =>
                        if conv_valid = '1' then
                                nr_state <= uart_and_control;
                                load_tx <= '1';
                                nr_data2tx <= r_data_conv_o;
                        else
                                nr_state <= conv_ASCII;
                                load_tx <= '0';
                                nr_data2tx <= r_data2tx;
                        end if;
                when send_bcd =>
                        if aux_cnt = 0 then
                                nr_control <= "0011"; -- next state is send_cab
                                nr_state <= conv_ASCII;
                                nr_data2conv <= r_12b (11 downto 8);
                                enable_conv <= '1';
                                substract_1 <= '0';
                        else -- aux_cnt > 0
                                nr_control <= "0010"; -- next state is send_bcd
                                nr_state <= conv_ASCII;
                                nr_data2conv <= r_12b (11 downto 8);
                                enable_conv <= '1';
                                substract_1 <= '1';
                        end if;
                when send_address => -- the address of the slave is the first Byte
                        if r_complete_byte = '0' then -- the 4 bits group is the beginning of the address byte
                                shift_4 <= '1';
                                nr_state <= conv_ASCII;
                                enable_conv <= '1';
                                nr_data2conv <= r_8b (7 downto 4);
-- next state should be send_address again to send the low part of the address byte
                                nr_control <= "0101";
                                nr_complete_byte <= '1'; -- the next 4 bits group will be the end of a Byte
                        else
                                nr_state <= conv_ASCII;
```

```
                        enable_conv <= '1';
                        nr_data2conv <= r_8b (7 downto 4);
-- next state should be send_cab to send the read or write character
                        nr_control <= "0110";
                        nr_complete_byte <= '0';
                end if;
        when send_byte =>
                if r_complete_byte = '0' then -- the 4 bits group is the beginning of the byte
                        shift_4 <= '1';
                        nr_state <= conv_ASCII;
                        enable_conv <= '1';
                        nr_data2conv <= r_8b (7 downto 4);
-- next state should be send_bye again to send the low part of the byte
                        nr_control <= "1010";
                        nr_complete_byte <= '1'; -- the next 4 bits group will be the end of a Byte
                else
                        nr_state <= conv_ASCII;
                        enable_conv <= '1';
                        nr_data2conv <= r_8b (7 downto 4);
                        nr_control <= "1011"; -- next state should be send_cab to send the acknowledge value
                        nr_complete_byte <= '0';
                end if;
        when send_ack =>
                if FIFO_dout_2(0) = '0' then -- the byte that is been sent isn´t the end of a message
                        nr_state <= conv_ASCII;
                        enable_conv <= '1';
                        nr_data2conv <= "000" & FIFO_dout_2(1); -- the acknowledge value is sent
                        nr_control <= "1000"; -- Next state will be send_cab to transmit another byte
                else -- the byte that is been sent is the end of a message
                        nr_state <= conv_ASCII;
                        enable_conv <= '1';
                        nr_data2conv <= "000" & FIFO_dout_2(1); -- the acknowledge value is sent
                        nr_control <= "1001"; -- Next state will be idle to wait another message
                end if;
        when others => -- uart_and_control
                if tx_busy = '0' then
                        case r_control is
                        when "0000" => -- Next state will be send_cab
                                add_pos <= '1';
                                rst_pos <= '0';
                                nr_state <= send_cab;
                                enable_bcd <= '0';
                        when "0001" => -- the message counter value should be sent
                                add_pos <= '0';
                                rst_pos <= '0';
                                enable_bcd <= '1';
-- The number of message is converted to BCD
                                nr_data2tx <= conv_std_logic_vector(msg_cnt,8);
-- the aux_counter is set to 2 (This counter changes its value to send hundreds, tens and units)
                                load_2 <= '1';
                                if r2_bcd_valid = '1' then -- wait until the conversion has finished
                                        nr_state <= send_bcd;
                                else
                                        nr_state <= uart_and_control;
                                end if;
-- hundreds or tens have been sent and another BCD character should be sent
                        when "0010" =>
                                shift_bcd <= '1';
                                add_pos <= '0';
                                rst_pos <= '0';
                                nr_state <= send_bcd;
                                enable_bcd <= '0';
```

```
-- the message counter value has been sent and we will send the address header
                                when "0011" =>
                                        nr_state <= send_cab;
                                        add_pos <= '1';
                                when "0100" => -- the high part of the address will be sent so we read FIFO memories
                                        if valid = '1' then -- FIFOS outputs could be read
                                                rd_en <= '0';
                                                nr_state <= send_address;
                                                load_byte <= '1';
                                        else    -- wait until the data could be read
                                                rd_en <= '1'; -- FIFO memories outputs are read
                                                nr_state <= uart_and_control;
                                        end if;
                                when "0101" => -- the low part of the address will be sent
                                        nr_state <= send_address;
                                when "0110" => -- the acknowledge header will be sent
                                        add_pos <= '1';
                                        nr_state <= send_cab;
                                when "0111" => -- the acknowledge value will be sent
                                        nr_state <= send_ack;
-- the ack value have been sent and the message contains another byte
                                when "1000" =>
                                        nr_state <= send_cab;
                                        bit_pos <= '1';
-- the ack value have been sent and the last byte was the end of a message
                                when "1001" =>
                                        nr_state <= uart_and_control;
                                        load_tx <= '1';
                                        if r_tx_end = '1' then
                                                nr_data2tx <= conv_std_logic_vector(13,8); -- Carriage return
                                                nr_control <= "1111"; -- end of transmission
                                                nr_tx_end <= '0';
                                        else
                                                nr_data2tx <= conv_std_logic_vector(10,8); -- End of line
                                        -- return to this state to send the carriage return
                                                nr_control <= "1001";
                                                nr_tx_end <= '1';
                                        end if;
                                when "1010" => -- the high part of the byte will be sent
                                        nr_state <= send_byte;
                                when "1011" => -- the byte has been sent completely and we will send the ack header
                                        nr_state <= send_cab;
                                        add_pos <= '1';
                                when "1100" => -- the high part of the byte will be sent so we read FIFO memories
                                        if valid = '1' then -- FIFOS outputs could be read
                                                rd_en <= '0';
                                                nr_state <= send_byte;
                                                load_byte <= '1';
                                        else    -- wait until the data could be read
                                                rd_en <= '1'; -- FIFO memories outputs are read
                                                nr_state <= uart_and_control;
                                        end if;
                                when others => -- end of a message
                                        nr_state <= idle;
                                        add_msg <= '1';
                                        rst_pos <= '1';
                                end case;
                        else
                                nr_state <= uart_and_control;
                        end if;
end case;
end process;
```

```
--OUTPUTS
LED_EMPTY <= empty;
LED_MSG_LOST <= lost_msg;
end Behavioral;
```

11.4.8 Conversor de Binario a BCD

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
entity BIN_TO_DEC_CONVERTER is
   Port ( CLK : in  STD_LOGIC;              -- Main clock input
          RST : in  STD_LOGIC;              -- Reset actived high port
-- 8 bits input (the module is ready to convert a vector of maximum 8 bits)
          DIN : in  STD_LOGIC_VECTOR (7 downto 0);
          ENABLE : in  STD_LOGIC;                                  -- Port to start a conversion
-- BCD equivalent value (8 bits have a maximum value of 255, it means a representation of 3 BCD characters that is
equivalent to 12 bits)
          DOUT : out  STD_LOGIC_VECTOR (11 downto 0);
          DOUT_VALID : out  STD_LOGIC);                            -- Signal to indicate that a conversion has finished
end BIN_TO_DEC_CONVERTER;

architecture Behavioral of BIN_TO_DEC_CONVERTER is
-- Signals
signal r_cnt : unsigned (2 downto 0); -- Counter to calculate the number of bit conversions made
signal nr_cnt : unsigned (2 downto 0); -- new counter value
signal bcd : unsigned (11 downto 0) := (others => '0'); -- Bits vector to calculate the converted value
signal r_din : std_logic_vector(7 downto 0) := (others => '0'); -- DIN input register
signal nr_din : std_logic_vector(7 downto 0) := (others => '0');
signal r_enable : std_logic := '0'; -- Enable input register
signal flag_conv : std_logic := '0'; -- Flag to request a bit conversion
signal shift_flag : std_logic := '0'; -- Flag to shift a bit position the shift register
signal r_end_conv : std_logic := '0'; -- Flag to indicate the end of a complete conversion
-- FSM signals
type state is (idle, get_data, shift_bcd, conversion, shift_din);          -- FSM states
signal r_state : state;
signal nr_state: state;
begin
reg_process:process (RST, CLK)
begin
if RST= '1' then
            r_state <= idle;
            r_enable <= '0';
            r_cnt <= (others=>'0');
            r_din <= (others=>'0');
elsif CLK'event and clk = '1' then
            r_enable <= ENABLE;
            r_state <= nr_state;
            r_cnt <= nr_cnt;
            r_din <= nr_din;
end if;
end process;

bcd_conv_process:process (RST, CLK)
begin
if RST= '1' then
            DOUT <= (others=>'0');
            bcd <= (others=>'0');
elsif CLK'event and clk = '1' then
            if flag_conv = '1' then -- Bit conversion requested
```

```
                if(bcd(3 downto 0) > 4) then --add 3 if BCD digit is greater than 4.
                        bcd(3 downto 0) <= bcd(3 downto 0) + 3;
                end if;
        if(bcd(7 downto 4) > 4) then --add 3 if BCD digit is greater than 4.
                bcd(7 downto 4) <= bcd(7 downto 4) + 3;
        end if;
        if(bcd(11 downto 8) > 4) then  --add 3 if BCD digit is greater than 4.
                bcd(11 downto 8) <= bcd(11 downto 8) + 3;
        end if;
        elsif shift_flag = '1' then -- a new bit of the Input is stored
                bcd <= bcd (10 downto 0) & r_din(7);
        else
                bcd <= bcd;
        end if;
        if r_end_conv = '1' then -- A complete transmission has been done
                DOUT <= conv_std_logic_vector (bcd,12);
                bcd <= (others=>'0');
        end if;
end if;
end process;

fsm_process: process (r_state, ENABLE, r_enable, DIN, r_cnt, r_din)
begin
-- Default values
nr_state <= r_state;
r_end_conv <= '0';
nr_din <= r_din;
nr_cnt <= r_cnt;
flag_conv <= '0';
shift_flag <= '0';
        case r_state is
        when idle =>
                r_end_conv <= '0';
                nr_cnt <= (others=>'0');
                flag_conv <= '0';
                shift_flag <= '0';
                if ENABLE = '1' and r_enable = '0' then -- start condition
                        nr_state <= get_data;
                end if;
        when get_data => -- The input value is stored
                r_end_conv <= '0';
                nr_din <= DIN;
                flag_conv <= '0';
                shift_flag <= '0';
                nr_state <= shift_bcd;
        when shift_bcd => -- A first shift is made. the first bit of the input is stored in the conversion register
                r_end_conv <= '0';
                flag_conv <= '0';
                shift_flag <= '1';
                nr_state <= conversion;
-- when 7 bit conversions have been made, we store the last Input bit in the conversion register (like is described in
the algorithm)
        when conversion =>
                if r_cnt < 7 then
                        nr_cnt <= r_cnt + 1;
                        nr_state <= shift_din;
                        r_end_conv <= '0';
                        flag_conv <= '1';
                        shift_flag <= '0';
                else -- after 7 bit conversions and when the last bit is stored, the conversion has finished
                        nr_state <= idle;
                        r_end_conv <= '1';
```

```
                        end if;
-- shift_din. The input register is shifted so we will have the next bit to convert in the highest position of the register
            when others =>
                        nr_state <= shift_bcd;
                        r_end_conv <= '0';
                        nr_din <= r_din(6 downto 0) & '0';
                        flag_conv <= '0';
                        shift_flag <= '0';
  end case;
end process;

-- Output assignment
DOUT_VALID <= r_end_conv;
end Behavioral;
```

11.4.9 Conversor de hexadecimal a ASCII

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_arith.all;
entity HEX_TO_ASCII_CONVERTER is
   Port ( CLK : in  STD_LOGIC; -- Main clock input
          RST : in  STD_LOGIC; -- Reset actived high port
          DIN : in  STD_LOGIC_VECTOR (3 downto 0); -- 4 bits input (hexadecimal character to convert)
          ENABLE : in  STD_LOGIC; -- Input to activate the conversor
          DOUT : out  STD_LOGIC_VECTOR (7 downto 0); -- Result of the conversion
          DOUT_VALID : out  STD_LOGIC); -- Output to indicate that the conversion has finished
end HEX_TO_ASCII_CONVERTER;

architecture Behavioral of HEX_TO_ASCII_CONVERTER is
signal init_conv: std_logic; -- Signal used to start a conversion
signal r_enable: std_logic; -- Enable input register
signal r_din: std_logic_vector(3 downto 0); -- Din input register

begin
reg_din_process:process (RST, CLK)
begin
if RST= '1' then
            r_din <= (others=>'0');
            init_conv <= '0';
            r_enable <= '0';
elsif CLK'event and clk = '1' then
            r_enable <= ENABLE;
            if r_enable = '1' then -- If a conversion is requested
                        r_din <= DIN; -- The value to convert is stored
                        init_conv <= '1'; -- The conversion is started
            else
                        r_din <= (others=>'0');
                        init_conv <= '0';
            end if;
end if;
end process;

conv_process: process (RST, CLK)
begin
if RST= '1' then
            DOUT <= (others=>'0');
            DOUT_VALID <= '0';
elsif CLK'event and clk = '1' then
            if init_conv = '1' then
                        DOUT_VALID <= '1'; -- new converted data is available
                        case r_din is
```

```
                when "0000" =>
                        DOUT <= conv_std_logic_vector(48,8); -- equivalent to 0x"0" in ASCII
                when "0001" =>
                        DOUT <= conv_std_logic_vector(49,8); -- equivalent to 0x"1" in ASCII
                when "0010" =>
                        DOUT <= conv_std_logic_vector(50,8); -- equivalent to 0x"2" in ASCII
                when "0011" =>
                        DOUT <= conv_std_logic_vector(51,8); -- equivalent to 0x"3" in ASCII
                when "0100" =>
                        DOUT <= conv_std_logic_vector(52,8); -- equivalent to 0x"4" in ASCII
                when "0101" =>
                        DOUT <= conv_std_logic_vector(53,8); -- equivalent to 0x"5" in ASCII
                when "0110" =>
                        DOUT <= conv_std_logic_vector(54,8); -- equivalent to 0x"6" in ASCII
                when "0111" =>
                        DOUT <= conv_std_logic_vector(55,8); -- equivalent to 0x"7" in ASCII
                when "1000" =>
                        DOUT <= conv_std_logic_vector(56,8); -- equivalent to 0x"8" in ASCII
                when "1001" =>
                        DOUT <= conv_std_logic_vector(57,8); -- equivalent to 0x"9" in ASCII
                when "1010" =>
                        DOUT <= conv_std_logic_vector(65,8); -- equivalent to 0x"A" in ASCII
                when "1011" =>
                        DOUT <= conv_std_logic_vector(66,8); -- equivalent to 0x"B" in ASCII
                when "1100" =>
                        DOUT <= conv_std_logic_vector(67,8); -- equivalent to 0x"C" in ASCII
                when "1101" =>
                        DOUT <= conv_std_logic_vector(68,8); -- equivalent to 0x"D" in ASCII
                when "1110" =>
                        DOUT <= conv_std_logic_vector(69,8); -- equivalent to 0x"E" in ASCII
                when "1111" =>
                        DOUT <= conv_std_logic_vector(70,8); -- equivalent to 0x"F" in ASCII
                when others =>
                        null;
                end case;
        else
                DOUT_VALID <= '0';
        end if;
end if;
end process;
end Behavioral;
```